Object-Oriented Analysis and Design of Concurrent, Real-Time Systems

KEVIN L. MILLS

INFT 860 SPRING 1993

DESIGN METHODS FOR
REAL-TIME SYSTEMS

GEORGE MASON UNIVERSITY

I. Introduction

As microprocessors fall in price and increase in performance, computing devices, and the software that controls such devices, assume a larger role in society. For example, many gasoline filling stations now offer computer-controlled pumps that allow a customer to specify his preference for octane, to fill his tank, and to pay for his purchase by credit card, all without human assistance. Automated teller machines (ATMs) abound in every shopping mall, grocery store, and airport terminal; credit card-activated telephones appear where travelers congregate; automobiles run more cleanly, efficiently, and safely with the aid of microprocessors; computer bulletin-board systems allow subscribers to scan product offerings, to select purchases, and to pay by credit card. All of these, increasingly common, computer applications exhibit some form of real-time processing and concurrency. Many involve distributed processing, as well. As the number and scope of these real-time, automated applications grow, our ability to analyze requirements and to design solutions for concurrent, real-time systems must improve. (Readers unfamiliar with the issues separating the design of concurrent, real-time systems from sequential applications should consult an introductory exposition, such as that provided by Laplante[1].)

Early attempts to analyze requirements for real-time systems focused on extensions to structured analysis. The resulting technique, Real-Time Structured Analysis (RTSA)[2], added control transforms, control event flows, and state transition diagrams to structured analysis. RTSA was first coupled with structured design to map real-time problems to sequential, one-task, designs. Later Research showed how RTSA could be mapped to a concurrent tasking design using Design Approach for Real-Time Systems (DARTS)[3]. For designing real-time systems, DARTS provides a significant improvement over structured design.

Introduction of Ada[†] as a programming language and run-time system for embedded, control software expanded the conceptual model through which most real-time, concurrent systems could be approached[4]. Ada included a multitasking model, synchronization techniques, and support for information hiding. These advances encouraged researchers to devise new methods for analyzing and designing real-time systems. One such method, concurrent object-based real-time analysis (COBRA)[5], evolved from RTSA. While retaining the notation from RTSA, COBRA adds: (1) guidelines for developing an environmental model, (2) guidelines for decomposing a system into subsystems, (3) criteria for identifying objects and functions, and (4) techniques for analyzing behavioral scenarios. COBRA also introduced the notion of aggregate objects into the analysis. COBRA analyses can be mapped readily into concurrent designs, and then into Ada implementations, using the Ada-based Design Approach for Real-Time Systems (ADARTS)[6].

Following the publication of Ada, an object-based programming language, further developments led to the emergence of object-oriented languages, such as C++[7] and Eiffel[8]. Object-oriented languages include expanded features for software reuse (inheritance, polymorphism, and object and method contracts)[9]. Object-oriented programming appears attractive because the cost of software development might be reduced as the amount of software reuse increases. As with Ada, object-oriented languages expanded the conceptual model available to software designers. Recently, a number of methods have emerged for analyzing and

† Ada is a registered trademark of the U.S. Department of Defense

designing systems using object-oriented concepts[10,11,12,13]. One such method, the object modeling technique (OMT)[12], appears to build on the concepts used in RTSA and COBRA, and, thus, might be applicable to concurrent, real-time systems.

This paper proposes that OMT provides a suitable analysis method for concurrent, real-time systems. The argument is supported through application of OMT to an example real-time application, an automated gas station management system (see Appendix A for the requirements statement). Further, a mapping is proposed from an OMT analysis to a design, based on an Object-Oriented Design Approach for Real-Time Systems (OODARTS), which extends the object-based ADARTS method to encompass a full, object-oriented, design model. To enable comparison between the RTSA, COBRA, and OMT methods, an analysis of the automated gas station management problem is presented using each of the methods. The RTSA specification is given in Appendix B, the COBRA specification in Appendix C, and the OMT specification in Appendix D. To facilitate an evaluation of the OODARTS design method against ADARTS, two designs for the gas station management system are shown. One design, included as Appendix E, uses the ADARTS method to design a solution from the COBRA analysis. The second design, included as Appendix F, uses the proposed OODARTS method to design a solution from the OMT analysis.

Before discussing the various analyses and designs shown in the appendices, this paper, in Section II, gives a brief description of the automated gas station problem. Section III discusses the various problem analyses: first, the RTSA analysis, followed by the COBRA specification and then the OMT model. Section IV presents the two design solutions: first, the ADARTS design, developed from the COBRA analysis, and then the OODARTS solution, developed from the OMT model. Section V provides a comparative evaluation of the strengths and weaknesses of the various analysis and design approaches. Special consideration is given to the applicability of OMT and OODARTS as analysis and design methods for concurrent, real-time systems. The paper closes with some conclusions and a list of references.

II. A Real-Time Gas Station Control Problem

The real-time problem used as an example in this paper should be familiar to many. Increasingly, gas stations are introducing automated pump processing to enable customers to purchase gasoline by inserting a credit card or a cash card, by selecting a type of gasoline, and then by pumping the gas for themselves. Of course, customers may still opt to pay in cash or by credit card at a booth where a human attendant waits. The attendant can also observe the gas station for safety hazards and emergencies, correcting hazards and reporting emergencies to the police and fire departments, as appropriate.

The gasoline station used as an example in this paper carries the familiar concept of an automation-assisted gas station to a future time where gas stations might be completely automated. Such stations would need automated gas station management (AGMS) software to control their operations. The reader should imagine that the Pal Sal (Pump A Little, Save A Lot), Inc. gas station chain is considering total automation of their gas stations. Further, image that Pal Sal has assembled a requirements statement for automating their gas stations. The imagined requirements statement is provided as Appendix A. To help to envision the problem, a conceptual diagram of an automated Pal Sal station is shown as Figure 1.

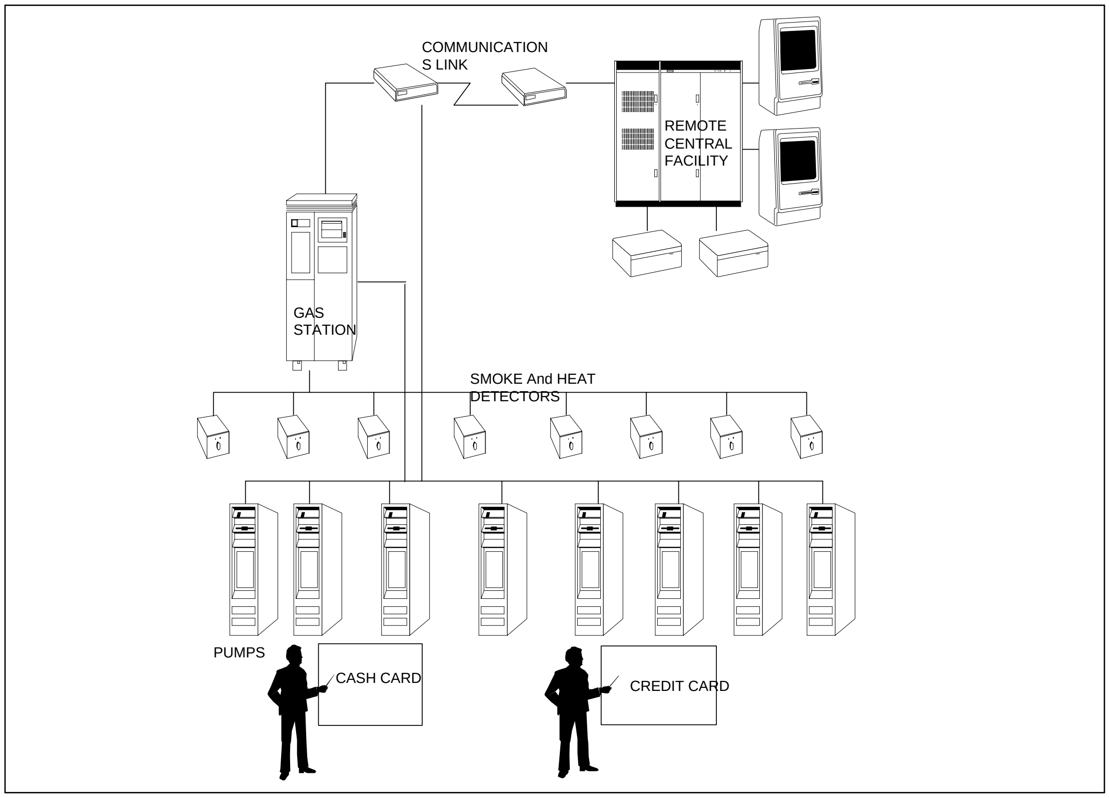


Figure 1. An Automated Gas Station Concept

Each gas station comprises a number of pumps, initially eight, capable of accepting credit or cash cards from customers, of dispensing gasoline, when authorized, of recording the transaction, and of updating a customer's cash card to deduct the amount of money used on gas. If a cash card is inserted, the pump can validate the card by examining the cash value; any customer can purchase gas with a cash card, up to the limit of cash on the card. If a credit card is inserted, the pump must send an authorization request to a remote central facility, and must then await a reply. If the card is validated, then gas can be dispensed until the customer turns off the switch, or the switch is shut off automatically. After processing a credit transaction, the pump sends the cost of the gas purchased, and the account number used to buy the gas, to the remote central facility. In the example, each pump is capable of dispensing only a single type of gasoline.

Each pump comprises a display to show the customer the amount and cost of the gas purchased, a switch to activate the pump dispenser, and two LEDs to inform the customer when their cash card is used or when their credit card is disapproved. Each pump operates autonomously, but a gas station controller can request that individual pumps finish the current transaction, if any, and then lock.

Each station includes a gas station controller that monitors safety and operating conditions at the gas station. Connected to the gas station controller, a set of paired smoke and heat detectors monitor the smoke particle concentration and temperature, respectively, at each pump. These detectors are illustrated in Figure 1 as eight boxes, one mounted at each pump. Each box contains a smoke and heat detector. Whenever a smoke particle concentration or temperature threshold is exceeded at one of the detectors, an alert is raised at the gas station controller. The gas station controller, upon receiving an alert, sounds an alarm at the station, sends an alarm message to the remote central facility, and then requests that each pump close operations. The gas station can also receive shutdown and restart requests from the remote central facility. Upon receiving a shutdown request, the gas station asks that all pumps cease operations. Upon receiving a restart request, the gas station asks that all pumps recommence operations.

A communications link, as shown in Figure 1, provides the path between the gas station controller and the remote central facility, as well as a means for the pumps and remote central facility to exchange information. The link can go up and down. When the link is down, the gas station is required to shutdown. When the link returns to operation, the gas station should automatically restart. When the link goes down, any transactions in progress are, of course, terminated, but a record of any credit transactions must be saved, so that the cost of gas purchased and the account number can be passed on to the remote central facility once the link resumes operation.

The remote central facility maintains a staff of operators to monitor alarms at each gas station and to maintain the central facility. The remote central facility also maintains a database of corporate credit accounts.

The interested reader should take a few minutes to scan the five page problem statement included as Appendix A. Those readers familiar with RTSA might also scan Appendix B to get a more precise understanding of the automated gas station management system (AGMS) requirements.

III. Problem Analyses and Specifications

At this stage in the paper, we switch writing styles to a more personal, first person plural. We believe this switch in viewpoints helps to emphasize the nature of the design process. Since the requirements have been stated, we begin to analyze those requirements, and to document our understanding. To effectively communicate our thinking, we believe that the reader should know that we exist, that we are thinking beings, and that we can make mistakes and reach assessments and possess opinions with which the reader may not agree. Also, we think that the more active writing style that flows naturally from a first person view will help us keep the reader’s attention.

In this, the third section of our paper, we present the analysis of the automated gas station management system (AGMS). We performed the analysis using three different methods, and we documented each of the analyses. For the first analysis we used Real-Time Structure Analysis (RTSA), the simplest, oldest, and most widely applied of the analysis techniques we present in this paper. Our main purpose in showing an RTSA specification is to present the AGMS in a more precise manner than the requirements statement in Appendix A by using an analysis method that many readers will be familiar with and that most readers, familiar or not, will be able to easily comprehend. The RTSA specification provides a level base for understanding the AGMS problem before we move on to consider progressively more complicated specifications resulting from the Concurrent Object-Based Real-time Analysis (COBRA) method and the Object Modeling Technique (OMT). So, we begin with RTSA.

A. RTSA -- A Great, Little Method for Real-Time Analysis

Real-Time Structured Analysis (RTSA) begins with the familiar concepts from structured analysis: context diagrams, data flow diagrams, data dictionaries, and puesdo-code specifications for leaf-level functions. To these concepts, the idea of control transforms, encompassing state transition diagrams is added. The result: a problem analysis method for real-time systems that is easy to use, easy to learn, and easy to understand. We have applied RTSA to our AGMS problem, and have documented the resulting specification in Appendix B.

The data and control flow diagrams for the AGMS are given on pages B-1 through B-7. The data dictionary comprises pages B-8 through B-11. The psuedo-code specifications, one for each leaf node data transform, are recorded on pages B-12 through B-17. The state transition diagrams, one for each control transform, are shown on pages B-18 through B-20.

The AGMS context diagram, B-2, delineates the boundary between the software and the hardware components of the AGMS. As you can readily see, the AGMS software must interact with a card reader, a communications link, some LEDs, some switches, an alarm, some gas dispensers, and some detectors. On page B-3, the AGMS is decomposed into three functions: 1) manage pump (one of which will exist for each pump in a gas station), 2) manage communications link (one per gas station), and 3) manage gas station (one per gas station).

B-4 presents a further decomposition of the manage pump function. Here we see that each pump monitors a pump on/off switch (1.2), and monitors a credit/cash card reader (1.3). These devices can cause events to which the pump must respond. The Control Pump control transform (1.1) accepts these events, as well as events arriving from the Manage Gas Station (3) and Manage Communications Link (2) transforms, analyzes the events against the current state of the pump, and then selects certain data transforms to activate. Data transforms controlled by Control Pump (1.1) include: Authorize Transaction (1.4), Dispense Gas (1.5), Complete Transaction (1.6), Reject Transaction (1.7), and Establish Transaction (1.8). All of the data transforms shown on B-4 are leaf-level transforms, and therefore, each of them has a corresponding mini-specification in psuedo-code. We encourage the reader to peruse these mini-specifications to get a better understanding of the AGMS requirements. The Control Pump control transform refers to a state transition diagram (on B-19) that defines the behavior of the transform. We hope the reader will review the finite state machine on B-19 to understand how Control Pump works. On the diagram, events are shown above lines, with corresponding actions shown below the same line. Conditions that must be satisfied coincident with an event are shown in [square brackets].

To run quickly through the Control Pump state diagram (see B-19), we see that the pump begins operation in the open state. Once a cash or credit card is inserted, the pump moves to the waiting authorization state, while invoking the Authorize Transaction function. If the transaction is authorized, the pump moves into the authorized state, unless the customer turned on the switch while authorization was pending, in which case the pump moves directly to the dispensing state and enables the Dispense Gas function. If the customer’s cash card runs out, then the gas dispenser stops, the Complete Transaction function is performed, and the pump returns to the open state. If the switch is turned off while dispensing, the pump tells the gas dispenser to halt and enters the waiting on done state. The left-hand side of the state diagram deals with requests to close the pump while the pump is in various states. We think the reader can now easily follow those events and actions. Once the pump enters the closed state, it remains there until an open event arrives from the Manage Gas Station data transform.

The Manage Communications Link (2) transform is further decomposed into two levels: B-5 shows the first level, and B-6 further decomposes the Send to Link (2.5) transform from B-5. The inputs to the Manage Communications Link transform are all allocated on B-5. Incoming messages from the communications link are handled by Add to Rcv List (2.1), which saves each message into the RCV LIST data store and then calls Decode Message Header (2.2). Decode Message Header removes a message from the RCV LIST data store, analyzes the message, and generates any events stimulated be the message. Incoming Link State Interrupts are handled by Analyze Link State (2.3). As appropriate, Analyze Link State will generate link events for Manage Gas Station (3) and will pass on any new link status to Send to Link (2.5). Add to Tx List handles messages flowing from Manage Gas Station (3) and from any of the Manage Pump (1) transforms. Add to Tx List saves the incoming message into the TX LIST data store and then sends a Wakeup to Send to Link (2.5). Send to Link is further decomposed on B-6.

Receiving a new link status or receiving a Wakeup causes Transmit Message (2.5.1) to act. On a Wakeup, Transmit Message checks the TX LIST for messages to send and checks the link status for an up link. If all conditions are ready, a message is removed from TX LIST and sent as an Outgoing Message. If a new link status changes the link from up to down, then Transmit Message (2.5.1) sends a Save to Save Credit Transactions (2.5.2) which then moves any credit transactions from the TX LIST to the CREDIT TRANSACTION LIST while discarding any other messages in TX LIST. If a new link status changes the link from down to up, then Transmit Message (2.5.1) sends a Restore to Restore Credit Transactions (2.5.3) which then moves the contents of CREDIT TRANSACTION LIST to the TX LIST.

The Manage Gas Station (3) transform is further decomposed on B-7. Here we see that Manage Gas Station is controlled by a state transition diagram (shown on B-20) embodied in Control Gas Station (3.1). The smoke and heat detectors in the gas station are monitored by Monitor Detectors (3.2) which generates a Threshold Exceeded event when appropriate. Other events arrive at Control Gas Station from Manage Communications Link (2). Control Gas Station can trigger any of five functions: Sound Alarm (3.3), Reset Alarm (3.4), Send Alarm Message (3.5), Send Opens (3.6), and Send Closes (3.7). The behavior of each of these functions is described in a mini-specification. The conditions under which each of these transforms is triggered are detailed in the Control Gas Station state transition diagram (B-20). We encourage the reader to review the Control Gas Station state transition diagram, and the related mini-specifications.

The data dictionary on B-9 through B-11 is self-explanatory. Each input and output data item shown on the context diagram is described. Also the internal data stores are described. From RTSA, we now take a step up in complexity to COBRA.

B. COBRA -- A Means To Analyze Larger Real-Time Systems

The Concurrent Object-Based Real-time Analysis (COBRA) method starts with RTSA as a base and adds several extensions intended to help analysts understand large, real-time systems. COBRA divides the analysis of a system into two parts: 1) an environmental model and 2) a behavioral model. The environmental model comprises a system context diagram, and where appropriate, supporting subsystem context diagrams. COBRA includes a set of guidelines for developing the environmental model. In fact, one of the extensions to RTSA provided by COBRA is the concept of decomposing a large, real-time system into subsystems. Another extension with COBRA permits representing subsystem components not only as functions, but also as objects (thus, the object-based component of the name COBRA). COBRA includes criteria for identifying functions and objects. Because COBRA permits representation of objects, the level of information hiding supported exceeds that available within RTSA. A final extension of note provided with the COBRA method is a technique for behavioral scenario analysis. Scenario analysis yields a more rigorous development of the necessary state transition diagrams than is possible using RTSA.

For the AGMS problem, our COBRA analysis is documented on pages C-1 through C-54. The system context diagram (C-2) mirrors that given in the RTSA specification; however, we immediately decompose the problem into three subsystems, each viewed as aggregate objects, illustrated on C-3. One subsystem, for real-time control, is a pump object (one instance of this object will exist for each pump in a given gas station), one is a communications (server) object, and the other is a gas station (real-time coordination) object. Here, each of the external data and control flows are allocated to one of the subsystems, and data and control flows between the objects are identified. The subsystem context diagrams on C-4, C-5, and C-6 represent each subsystem's context, showing other subsystems, when inter-subsystem data and control flows exist, as terminators. This completes our COBRA environmental model for the AGMS. Next, we developed the behavioral model.

We developed one behavioral model for each of the three subsystems we identified in our environmental model. A COBRA behavioral model consists of: 1) data/control flow diagrams, 2) a data dictionary, 3) psuedo-code for each leaf-level data transform, and 4) a state transition diagram for each control transform. In addition, the scenario analysis supporting the development of the state transition diagrams becomes part of the behavioral model. We included the scenario analysis apart from a specific subsystem because we used the scenario analysis to verify and correct our state transition diagrams. Thus, we viewed the scenario analysis as a system-level specification, rather than a subsystem-level specification.

Our behavioral model for the pump subsystem can be viewed on pages C-8 through C-16. C-9 provides the top-level view of the subsystem through a data/control flow diagram. Here, we decompose the pump subsystem into five objects: a Pump Control object (1.1), a Switch (1.2), a Card Reader (1.3), LEDs (1.4), and a Gas Dispenser (1.5). Each of these, except the Pump Control object, are leaf objects, and, so, a psuedo-code specification for each is included with the mini-specification section (pages C-13 through C-15). We further decompose the Pump Control object on C-10 into a Control Pump control object (1.1.1) and four supporting functions: Authorize Transaction (1.1.2), Establish Transaction (1.1.3), Complete Transaction (1.1.4), and Reject Transaction (1.1.5). Each of these leaf-level data transforms is further specified with puesdo-code. We further specify the control object with a state transition diagram show on C-16. We document data items for the pump subsystem on pages C-11 and C-12.

Our behavioral model for the communications subsystem can be seen on pages C-20 through C-26. (Somehow, we managed to include two pages numbered C-26. Here, we refer to the first of these.) C-21 provides the top-level view of the subsystem through a data flow diagram. We included no control objects in the communications subsystem. We further decompose the Send to Link function (2.5) on C-22. Thus, we decomposed the communications subsystem into seven leaf-level data transforms, or functions: Add to Rcv List (2.1), Decode Message Header (2.2), Analyze Link State (2.3), Add to Tx List (2.4), Transmit Message (2.5.1), Save Credit Transactions (2.5.2), and Restore Credit Transactions (2.5.3). We provide puesdo-code for each of these functions on pages C-23 and C-24. We give a data dictionary for the communications subsystem on pages C-25 and C-26 (the first one).

Our behavioral model for the gas station subsystem is exhibited on pages C-26 (the second of the C-26's) through C-30. C-27 shows the entire subsystem on one data/control flow diagram, including three objects and three functions. We specify the Control Gas Station control object (3.1), and its three supporting functions (Send Alarm Message (3.4), Send Opens (3.5), and Send Closes (3.6)), with a state transition diagram (C-30) and psuedo-code specifications (C-29), respectively. Our subsystem design also includes two device objects: Detector Array (3.2) and Alarm (3.3). We further specify these objects with psuedo-code on C-29. We include a data dictionary for the gas station subsystem on C-28.

The final part of our COBRA analysis for the AGMS entails an analysis of various behavioral scenarios, as documented on pages C-31 through C-52. We used these scenarios to verify our two state transition diagrams, as shown on pages C-53 and C-54. We describe each scenario that results from an external event (C-31-1 through C-31-4), we show how each scenario flows through our AGMS (C-32 through C-52), and then we relate each scenario to our two state transition diagrams (C-53 and C-54).

From our COBRA model, we now move on to examine a method that relies on objects and object-oriented concepts as the basis for problem analysis. We continue to use the AGMS as our problem statement.

C. OMT -- Objects Most Telling, Objects Most Timeless

The Object Modeling Technique (OMT) represents a problem from an object-oriented point of view, but uses three sub-models to do so. The *object model* provides the fundamental view of the problem. The *object model* describes the structure of objects within a problem, the relationships between the objects, the attributes of each object, and ultimately, the functions, or operations, of each object. The *dynamic model* yields a description of those aspects of a problem that deal with issues of timing, sequencing, and control. The *functional model* represents those aspects of a problem that require transformations of values, independent of when those transformations occur. These three models, object, dynamic, and functional, are to be viewed as related, with the *object model* being fundamental. The *object model* provides the main, integrating view of a problem analysis. The operations in the *object model* correspond to events in the *dynamic model* and functions in the *functional model*. The *dynamic model* describes control regimes for objects that require such. The *functional model* contains functions that are invoked through object operations or through actions in the *dynamic model*. Functions will operate on attributes within the *object model*. The *functional model* might also describe constraints on various object attributes.

The major motivation for OMT appears to be the philosophy that object modeling yields a view that will be easiest to understand by the customer and the analyst, designer, and programmer, and that object models provide a fundamental, problem-oriented, structure more likely to endure the vicissitudes of changing requirements. For this reason, we have dubbed the OMT acronym with two alternate specifications: Objects Most Telling (easier to understand) and Objects Most Timeless (more likely to endure).

For the AGMS problem, our OMT specification comprises pages D-1 through D-54. Our Object Model for the AGMS, pages D-1 through D-9a, includes: a basic object model (sans operations) (D-2), an object dictionary (D-3 through D-9), and a complete object model (including operations) (D-9a). We will present a description, in a little while, of how we developed this object model. Suffice to say for now that our preliminary object model and a draft of the object dictionary was developed first, and that a complete object model and final object dictionary was produced only after the dynamic and functional modeling were completed.

Our dynamic model encompasses the five state charts shown on pages D-10 through D-15. In due course, we will discuss how this model was developed.

Our functional modeling warrants some discussion. We originally attempted to develop a functional model following the guidelines presented by Rumbaugh, et al.[12] The results of our efforts are documented on pages D-16 through D-34. These pages begin with a system context diagram (D-17) and progress through several levels of data flow diagram decomposition (D-19 through D-26). We were most unhappy with the outcome of this exercise. Many of the data transformations are stimulated by unseen control transforms (the functional model is restricted to data transforms). Another source of concern is the intent of the functional model. The functional model is to help us define operations on objects we defined in the preliminary object model, and yet, not objects are included in the functional model. Despite our misgivings, we completed the functional model with psuedo-code, function descriptions for leaf-level data transforms (D-27 through D-31) and with a data dictionary for the functional model (D-32 through D-34).

Because we found our functional model to be confusing and, potentially, of little use for identifying operations in our object model, we developed an alternate functional model (pages D-35 through D-54). We began our alternate model with the same context diagram (D-17, repeated as D-36) with which we started our original functional model. From the context model, we decomposed the AGMS into a set of objects (D-37) using our preliminary object model as a guide. The initial decomposition, into an object communications diagram, includes two aggregate objects (Pump and Gas Station) and one leaf-level object (Communications Link). Here we allocated external data and control flows to the objects, and we identified, using mainly the dynamic model, some event flows between object. In this model, event flows are viewed as function flows where one object calls a function in another. For example, the AUTHORIZED flow from the Communications Link object to the Pump object is viewed as if the Communications Link is calling an AUTHORIZED function within the Pump.

The Gas Station aggregate object is further decomposed, as shown in the object communications diagram on D-38, into four leaf-level objects: Smoke Detector, Heat Detector, Gas Station, and Alarm. Again, these objects come from the preliminary object model.

The Pump aggregate object is decomposed, as shown in the object communications diagram on D-39, into nine leaf-level objects: Card Reader, Cash Card, Credit Card, Cash Transaction, Credit Transaction, LED, Gas Dispenser, Switch, and Pump. These objects also appear on the preliminary object model.

Once we defined the hierarchy of objects as a set of object communications diagrams OCDs), we transferred events from the dynamic model to the OCDs. This helped us verify the flow of events through the object model and forced us to specify the inter-object communications requirements of the AGMS. From here, we used the attributes in the preliminary object model to document which objects get and set the attributes. We placed these function flows on the OCDs. Finally, we iterated over the OCDs, using the dynamic model and the preliminary object model as our guide, to identify additional inter-object function flows that are needed. Then, we transcribed those flows onto the OCDs.

After completing the analysis of our OCDs, we identified those objects that required a more detailed functional model. For each such object, we created an object function diagram (OFD). Our OFDs for the AGMS are shown on D-40 through D-46. Calls to the data transforms on the OFDs can be traced from the OCDs (except where specific calls are made from within an object), but the OFD approach truly allows the specification of an object's functions independent of the specific context in which the object will be deployed. This independence is a key advantage of object-oriented modeling.

The OFD, our own invention, coupled with the OCD, enabled us to tie the functional model to the object and dynamic models. The logical coherence we achieved enabled us to produce a functional model consistent with the object view of OMT. Functional modeling as defined by Rumbaugh, et al.[12], we found to be divorced from the object model. This separation leads easily to a logical incoherence between the object, dynamic, and functional views of an OMT model. Such logical incoherence creates difficulties when allocating functions to the object model. We found that our use of OCDs and OFDs made for an easy allocation of functions to objects.

To complete our own functional model of the AGMS, we created object function descriptions (D-47 through D-54) for each data transform identified on an OFD. Each function description is shown using psuedo-code. We then updated our object dictionary to include a specification of the calling sequence for each function from our function descriptions. Finally, we updated our preliminary object model (D-2) to include the allocation of functions to object operations, thus producing our complete object model for the AGMS (D-9a).

Now we propose to describe the process we used to create our OMT model of the AGMS. The reader who is uninterested in this discussion can certainly move ahead to Section IV where we describe two alternate designs for a solution to the AGMS problem. One design begins with our COBRA specification and uses ADARTS to develop a solution. The other design begins with our OMT specification and uses OODARTS to develop a solution. The interested reader can press ahead to understand how we created our OMT model.

Our first job was development of the preliminary object model from the problem statement. We employed a set of steps provided by Rumbaugh, et al.[12]

1. Identify candidate objects.
2. Discard inappropriate objects.
3. Initiate a data dictionary. (**We enlarged this to become an object dictionary**.)
4. Identify candidate associations.
5. Discard inappropriate associations.
6. Identify candidate attributes.
7. Discard inappropriate attributes.
8. Refine inheritance.
9. Test access paths through the model.
10. Iterate on 1 through 9. (**This iteration is continuous during dynamic and functional modeling as well**.)

The Rumbaugh book provides many suggestions for how to carry out each of these steps. We found the suggestions given quite useful. We were able to develop a reasonable object model for the AGMS after reading the Rumbaugh book, and without ever having developed previously an object-oriented model. We had the greatest difficulty with the identification of relationships. We kept trying to specify functional relationships, rather than structural relationships. Once we overcame this impediment, our modeling proceeded smoothly. The results we obtained from following these steps are shown below as Figure 2.

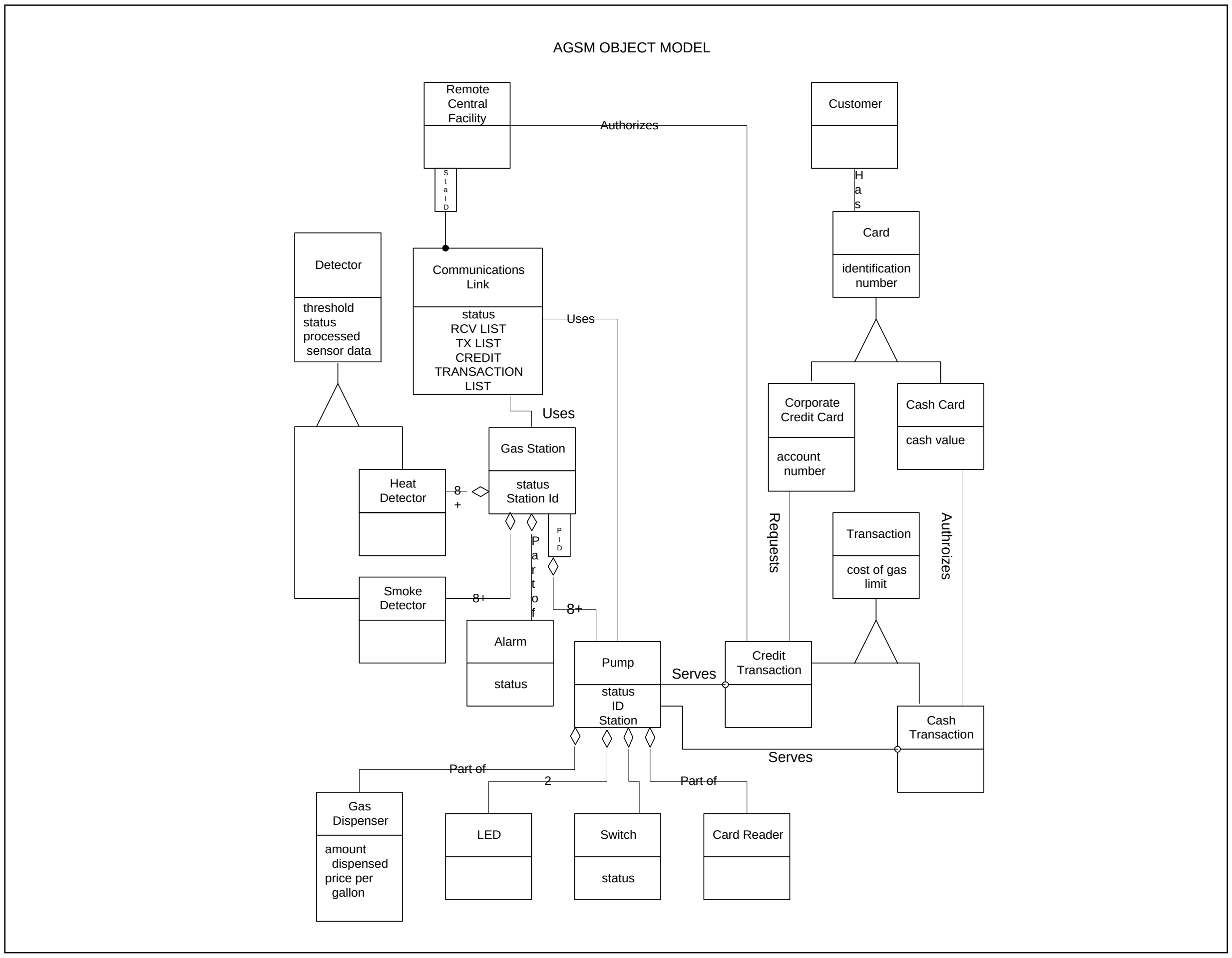


Figure 2. Preliminary AGMS Object Model

We next focused on creating the dynamic model. We began by following the procedure recommended in the Rumbaugh book.

1. Prepare scenarios.
2. Simulate the user interface.
3. Identify events.
4. Build state charts.
5. Match events with objects.

We defined scenarios by identifying external events that arrived into the AGMS. We categorized these events by source and destination. We envisioned a dynamic entity as the destination for each event. This immediately led us to define state charts for the communications link, for the gas station, and for the switch. Upon iteration, we identified dynamic objects that would create internal events, and we discovered the destination of those events. This process led us to define a state chart for the pump. We identified the need for a state chart in the detectors only after considering the mechanism they would use to detect emergency conditions. Here we decided that each detector should only send one internal event for each time the detector found that the threshold had been crossed in an upward direction. This requirement could best be implemented with a state chart.

Before going on, we would like to make some comments on state charts, since they provide a more extensive set of semantics than the state transition diagrams used in RTSA and COBRA. We will key our comments to the state chart for the pump as shown in Figure 3.

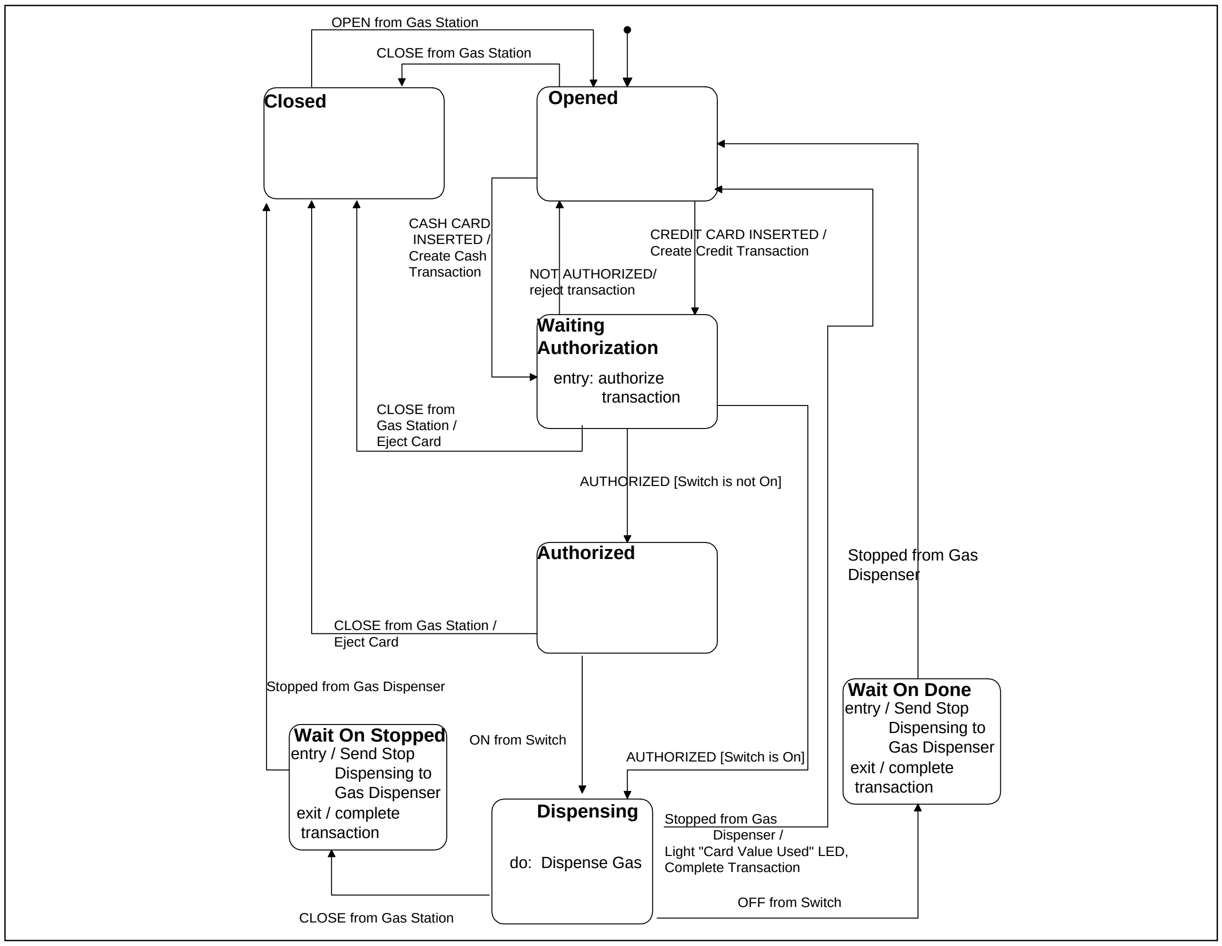

OPEN from Gas Station
CLOSE from Gas Station
Closed
Opened
CASH CARD INSERTED / Create Cash Transaction
CREDIT CARD INSERTED / Create Credit Transaction
NOT AUTHORIZED/ reject transaction
Waiting Authorization
entry: authorize transaction
CLOSE from Gas Station / Eject Card
AUTHORIZED [Switch is not On]
Authorized
Stopped from Gas Dispenser
CLOSE from Gas Station / Eject Card
Stopped from Gas Dispenser
Wait On Done
entry / Send Stop Dispensing to Gas Dispenser
exit / complete transaction
Wait On Stopped
entry / Send Stop Dispensing to Gas Dispenser
exit / complete transaction
ON from Switch
AUTHORIZED [Switch is On]
Dispensing
do: Dispense Gas
Stopped from Gas Dispenser / Light "Card Value Used" LED, Complete Transaction
OFF from Switch
CLOSE from Gas Station

Figure 3. Pump State Chart From OMT Dynamic Model Of AGMS

State chart notation represents states as rectangles with rounded corners. The state names are written in **bold** in the upper left-hand corner of a state. This provides room to annotate the state with three semantic devices: 1) a set of actions (following entry/) taken anytime a state is entered, 2) a set of actions (following do:) performed continuously while in a state, and 3) a set of actions (following exit/) taken upon leaving a state. The entry/ and exit/ notations enable the analyst to avoid repeating identical actions on all transitions into and out of a state, and the do: notation replaces the enable/disable operations on RTSA and COBRA state transition diagrams. Of course, each transition into and out of a state can also be labeled with actions taken only during that transition. The precedent rules are as expected: first, the actions on the specific transition into a state (see the **Waiting Authorization** state in Figure 3) are executed, followed by those associated with an entry/ (see, again, the **Waiting Authorization** state) notation in a state. Then, while in the state, all do: actions (see the **Dispensing** state in Figure 3) are continuously performed. Upon leaving a state, first the actions associated with the exit/ (see the **Wait On Stopped** state in Figure 3) notation in the state are performed, followed by any actions (see the STOPPED from Gas Dispenser transition from the **Dispensing** state in Figure 3) on the specific transitions out of the state.

Each transition is activated by a specific named event with the NAME in caps (see, for example, the CREDIT CARD INSERTED event on a transition leaving the **Opened** state in Figure 3). Optionally, the name can be modified by a from object, (for example, as in the CLOSE from Gas Station event on the transition leaving the **Opened** state in Figure 3) denoting the name of the object that sent or caused the event. An additional modifier can constrain a transition by placing a condition on the named event. Such conditions follow the event name, and the optional from modifier, and are enclosed in [square brackets] (see, for example, either AUTHORIZED event that causes a transition from the **Waiting Authorization** state in Figure 3). When actions occur during a transition, they are listed after a / that follows the event name and modifiers (for example, see the CLOSE from Gas Station event that causes an Eject Card action during the transition between the **Waiting Authorization** and **Closed** states in Figure 3).

These additional notational conveniences of state charts provide only superficial enhancements to state transition diagrams; however, the more regular notation of state charts probably improves the potential for automated tool support. An additional conceptual improvement that state charts do provide, however, is the ability to hierarchically nest and to sequentially decouple finite state machines. In the AGMS problem we did not use these features of state charts. The interested reader is referred to the Rumbaugh book, and to references there to work by Harel, for a fuller accounting of state chart notation.

Once the dynamic model was "completed" (please be aware that here, as with the object model, significant and continuous iteration is required), we developed the functional model, beginning with a system context diagram. We previously discussed the process we followed to decompose the system into Object Communication Diagrams and then to Object Function Diagrams and finally to Object Function Descriptions. Once the dynamic and functional models were complete, we began to add operations to our preliminary object model.

The Rumbaugh book suggests several sources for identifying object operations. We recount those sources here.

1. From the object model (i.e., gets and sets on attributes).
2. From events in the dynamic model.
3. From transition and state actions and activities (i.e., do: actions).
4. From functions within the functional model.
5. From shopping lists (i.e., a good object of this type should have these functions).

An initial decision we faced concerned events passed between the state charts. We could represent each event arriving at an object as an operation of that object, or we could choose to encapsulate the entire state chart within a single operation of the receiving object. We chose to encapsulate the state chart in a single operation. Usually, we denoted this operation by the name process_event. A second decision we faced concerning the state charts was whether to represent the generation of events with specific send operations, or to hide event generation within the process_event encapsulation operation. We chose to represent event generation explicitly.

After generating operations from the dynamic model, we turned our attention to the functional model. Here, we created operations for each function identified in the functional model for each object. This work was greatly simplified because our functional model was created within the context of our preliminary object model. We then added any operations that supported the internal operation of specific objects, but which seemed perhaps useful in some contexts when accessed from outside the object. This was a limited application of the shopping list suggested by Rumbaugh.

We had to step somewhat beyond the guidance provided in the Rumbaugh book if we were to define operations to support modeling of polymorphism, coupled with inheritance. Polymorphism appears in the Rumbaugh book on three pages. On page 2, Rumbaugh, et al., explain that polymorphism means that the same operation may behave differently in different classes. They fail to inform the reader that polymorphism can be combined with inheritance to produce some powerful results. On page 25, they explain that when classes share polymorphic operations each definition of the operation must have the same number and type of input parameters and the same return result, as well as a similar intent. Finally, on page 328, they mention polymorphism in connection with a discussion of the CLOS language. The limited discussion of polymorphism contained in the Rumbaugh book is simply one example of what John Palmer terms gross concept neglect.[14]

John Palmer explains that almost nothing is discussed about the concept of polymorphism in any of the most popular object-oriented analysis and design works, including Coad-Yourdon[13], Shlaer-Mellor[11], and Rumbaugh[12]. Palmer goes on to explain why polymorphism is an important object-oriented analysis and design concept. He gives four reasons.

1. Polymorphism can reduce the complexity of operation functional specifications.
2. Inclusion of polymorphism in object-oriented analysis can reduce the conceptual gap between analysis and object-oriented design.
3. Polymorphism is consistent with the way in which traditional analysis methods have evolved (i.e., precisely defined functions have been identified over time and have become standards for analysis).
4. Polymorphism is a concept that parallels the way users typically think (i.e., people often think of the same function carried out differently depending on the context).

Because we intended to take our OMT specification on to an object-oriented design, we decided that polymorphism should be employed in the analysis. Fortunately, OMT provides for abstract operations, and, indirectly, for redefining operations that have been given default behavior in a superclass.

Recall from the preliminary object model (Figure 2 or page D-2) that we identified three cases of inheritance: 1) Detector (inherited by Heat Detector and Smoke Detector), 2) Transaction (inherited by Credit Transaction and Cash Transaction), and 3) Card (inherited by

Corporate Credit Card and Cash Card). In each of these cases we employed polymorphism. At this stage we will introduce the complete object model, shown as Figure 4.

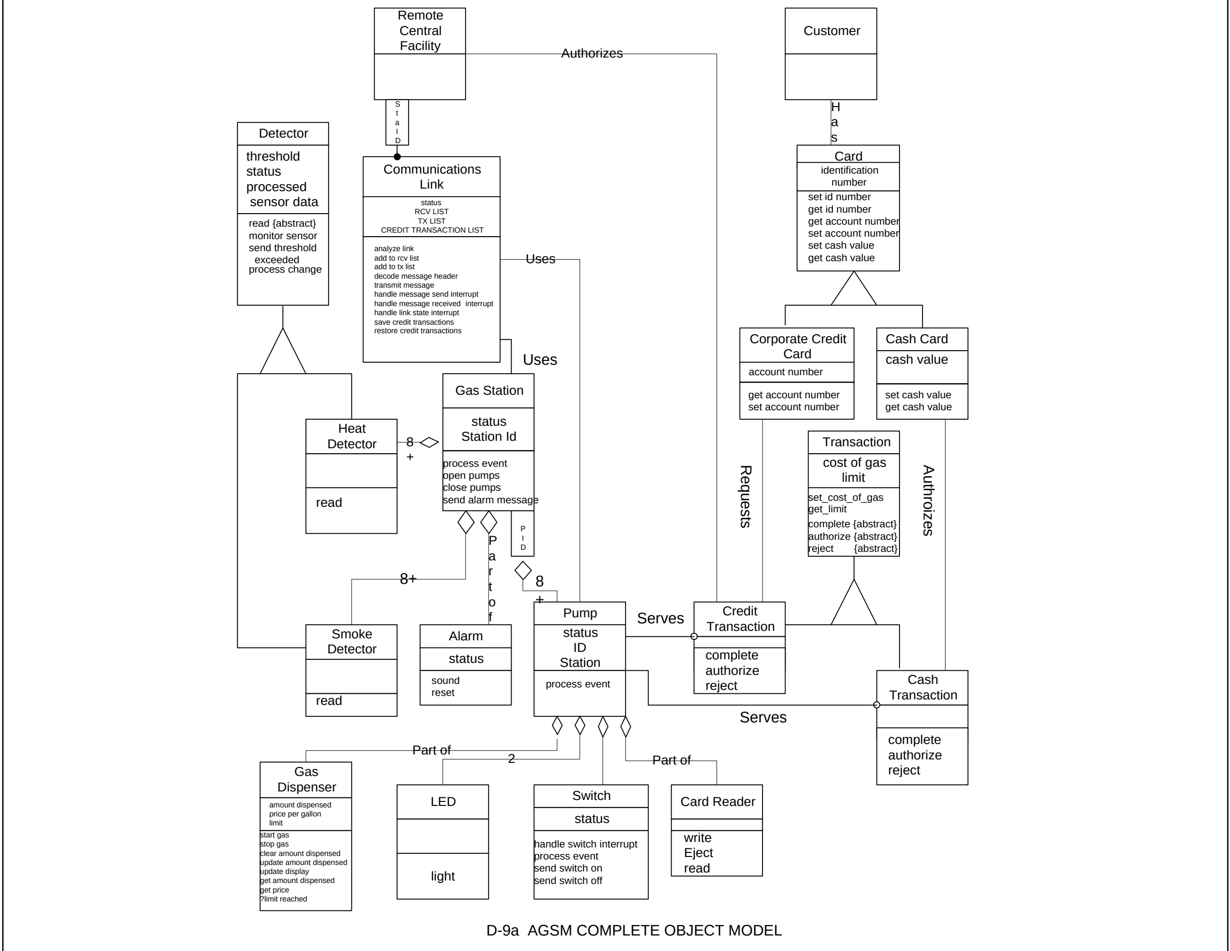


Figure 4. Complete Object Model for the AGMS Problem

In the case of the Detector superclass, we defined an abstract operation, read. (An abstract, or virtual, or deferred operation is one that defines what, but not how. No class containing an abstract operation may ever be instantiated, but such a class can be inherited.) In essence, the Detector class embodies all of the behavior of a detector except for the read operation. A read operation must be defined by any class that inherits the Detector class. Thus, in our specification of the AGMS problem, a Heat Detector and a Smoke Detector behave identically, except for how they read the sensor data. We employed a similar approach when defining the Transaction class.

The Transaction class contains three abstract operations: 1) complete, 2) authorize, and 3) reject. The behavior of each of these operations is defined differently for each of the subclasses of Transaction (i.e., Credit Transaction and Cash Transaction). Now, whenever a complete, authorize, or reject operation is invoked on a Transaction object, the correct behavior will occur for the specific type of Transaction that is referenced. For the complete and authorize operations, a Card class is used as an input parameter. This leads us to our final use of polymorphism in the OMT analysis of the AGMS.

The Card class, although meant to be inherited, can be instantiated because we have defined default behaviors for every operation of the class. The default behaviors give a logically consistent, although not very useful, set of operations for the Card class. The Cash Card class inherits Card, and redefines the set cash value and get cash value operations. In an object-oriented design, the set account number and get account number operations might be suppressed in the definition of a Cash Card, but in OMT no notation permits such suppression; therefore, we chose to let the default operations obtain when no redefinition is given in a subclass. Since the default operations were defined to work properly, no problem will occur.

The Corporate Credit Card class inherits Card, and redefines the set account number and get account number operations. The default definitions for get cash value and set cash value stand in this case.

After we identified all the operations needed for our AGMS object model, we documented the operation behaviors, using psuedo-code, on pages D-47 through D-54. We then updated the object dictionary (D-3 through D-9) to reflect the specification of the signature for each operation, and, finally, we produced a complete object model (D-9a) by adding to the preliminary object model the operations associated with each object class. This completes our OMT specification for the AGMS problem.

IV. Concurrent Design Solutions

In this section of the paper, we present two design solutions for the AGMS problem. One solution, included as Appendix E, begins with our COBRA specification and applies the procedure and notation from the ADARTS method.[6] The second solution, included as Appendix F, begins with our OMT specification and applies an Object-Oriented Design Approach for concurrent, Real-Time Systems (OODARTS). Rumbaugh, et al., provide very little guidance for creating a concurrent, real-time design from an OMT specification. (The skeptical reader is referred to Chapter 9 of the Rumbaugh book.[12]) ADARTS provides a good method and notation for design of concurrent systems, but does not assume a strictly object-oriented environment. OODARTS was devised by us to facilitate the generation of an object-oriented design from an OMT specification. We derived OODARTS from the solid foundation laid by the ADARTS method and notation. In most cases, we adopted ADARTS techniques with only slight adaptations, as required for an object-oriented environment. We will explain our approach in due course, but first we describe our ADARTS design.

A. ADARTS -- A Recipe for Solutions

ADARTS can be thought of as a recipe for concurrent, real-time solutions because, like a recipe, ADARTS delineates steps for producing a dish (the design) from a set of ingredients (the products of an analysis, such as RTSA or COBRA). Also like a recipe, the delectability of the dish varies with the skill of the chef, yet even an inexperienced cook can produce an edible dish simply by following the steps provided.

ADARTS provides a three-phase approach to system design. In phase one, tasks are identified, using a set of criteria for task structuring, and a task architecture is created. An overview of our ADARTS task architecture for the AGMS is given on pages E-1 through E-5. By applying the ADARTS task structuring criteria to the COBRA data/control flow diagrams in Appendix C, we created the task architecture shown on E-3, and reproduced below as Figure 5. The architecture is repeated on E-4, where the inter-task message flows are annotated with numbers to facilitate a discussion of the message flows through the system of tasks.

For each task identified, ADARTS requires that a task behavior specification (TBS) be produced. For our AGMS design, the task behavior specifications are given on pages E-6 through E-27. Each TBS: 1) names the task, 2) describes the input and outputs, categorized as events, messages, and data, of the task, 3) identifies references to any information hiding modules (IHMs, as discussed below), 4) specifies the ADARTS criteria used to identify the task and provides references to the control and data transformations in the problem analysis that are included within the task, 5) indicates the tasks timing characteristics and priority, 6) specifies the thread of control for the task, and 7) documents any errors that the task detects or avoids.

The TBSs are produced initially from the problem analysis, and then, in the third phase of the ADARTS method, are updated to include references to, and operations from, information hiding module (IHM) specifications, the product of the second phase of the ADARTS method. Our IHM specifications for the AGMS design are shown on pages E-28 through E-38. IHMs are identified using a set of module structuring criteria included within the ADARTS method. For each IHM in the design, ADARTS requires a specification that: 1) names the module, 2) describes the information hidden within the module, 3) identifies the ADARTS structuring criteria used to define the module, 4) documents any assumptions made about the module, 5) anticipates any changes that will be made to the module, and 6) specifies each operation included in the module.

Once the IHMs are specified, ADARTS requires that a system architecture be produced by allocating modules to tasks. This system architecture design comprises the third phase of the ADARTS method. Also, during this phase of the design, ADARTS requires that the TBSs be updated to account for the IHMs. This amounts to including IHMs in the appropriate reference section of each TBS and to adding IHM operation calls to the TBS thread of control section, when appropriate. Our ADARTS system architecture for the AGMS problem is shown on pages E-39 through E-41. We can now run quickly through the ADARTS design we produced from the COBRA specification of the AGMS problem.

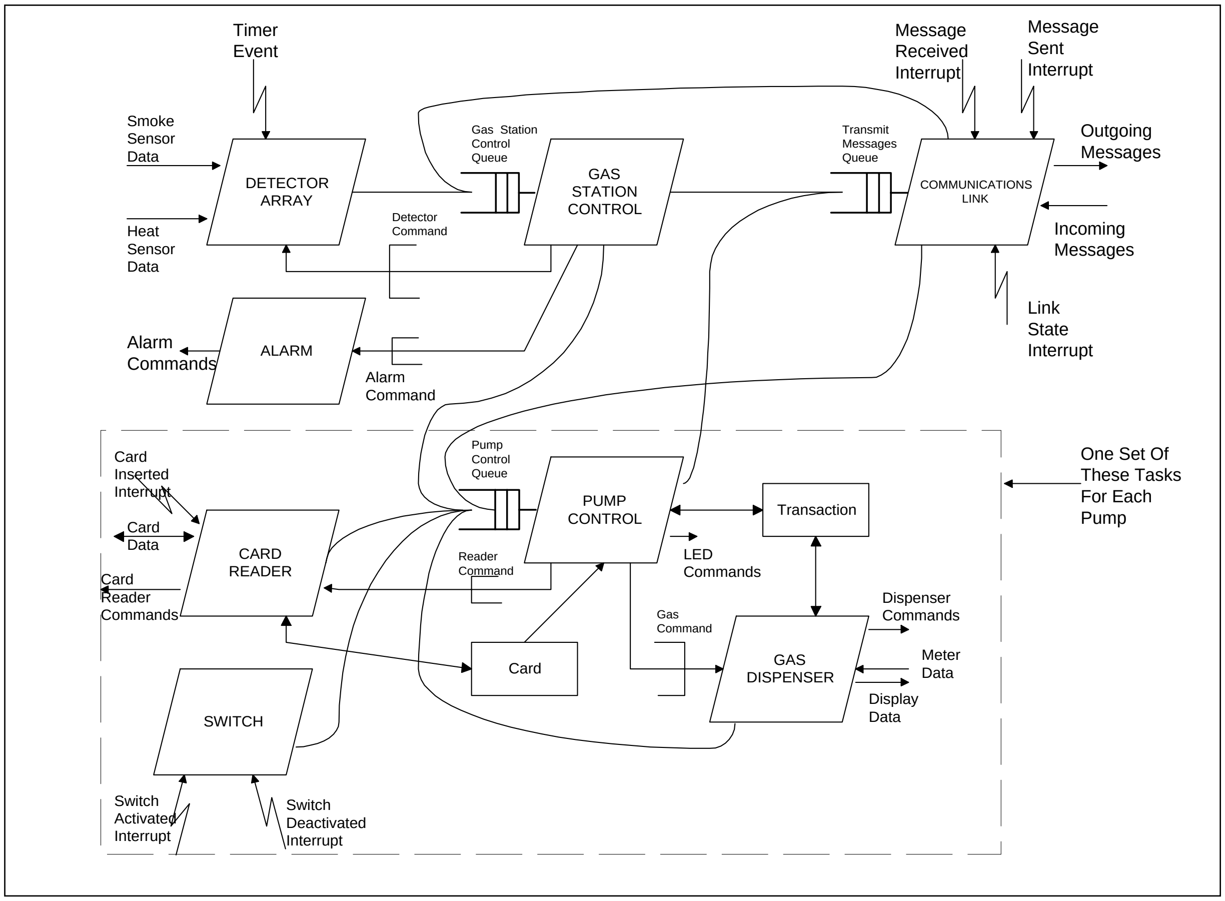


Figure 5. ADARTS Task Architecture for the AGMS from a COBRA Specification

As shown in Figure 5, three tasks comprise the gas station (Detector Array, Alarm and Gas Station Control), one task provides communications services, and a set of four tasks control each pump within a gas station. The Card IHM is shared between the Card Reader and Pump Control tasks, and the Transaction IHM is shared between the Pump Control and Gas Dispenser tasks.

The reader can probably discern that ADARTS shows tasks as parallelograms and IHMs as rectangles. Message flows, shown as arcs between tasks, can be loosely-coupled (flowing into queues) or tightly-coupled, without reply (flowing through half rectangles) or with reply (flowing through rectangles that have been flayed and twisted -- note that no icons for tightly-coupled messages with reply are shown in Figure 5). External events are shown as jagged lines with arrowheads attached. For a detailed accounting of the ADARTS notation, the reader is referred to the creator of ADARTS.[6]

After we identified the IHMs in our design, we allocated those IHMs to the AGMS tasks to create a system architecture diagram, as shown below in Figure 6. We will describe the allocations we made.

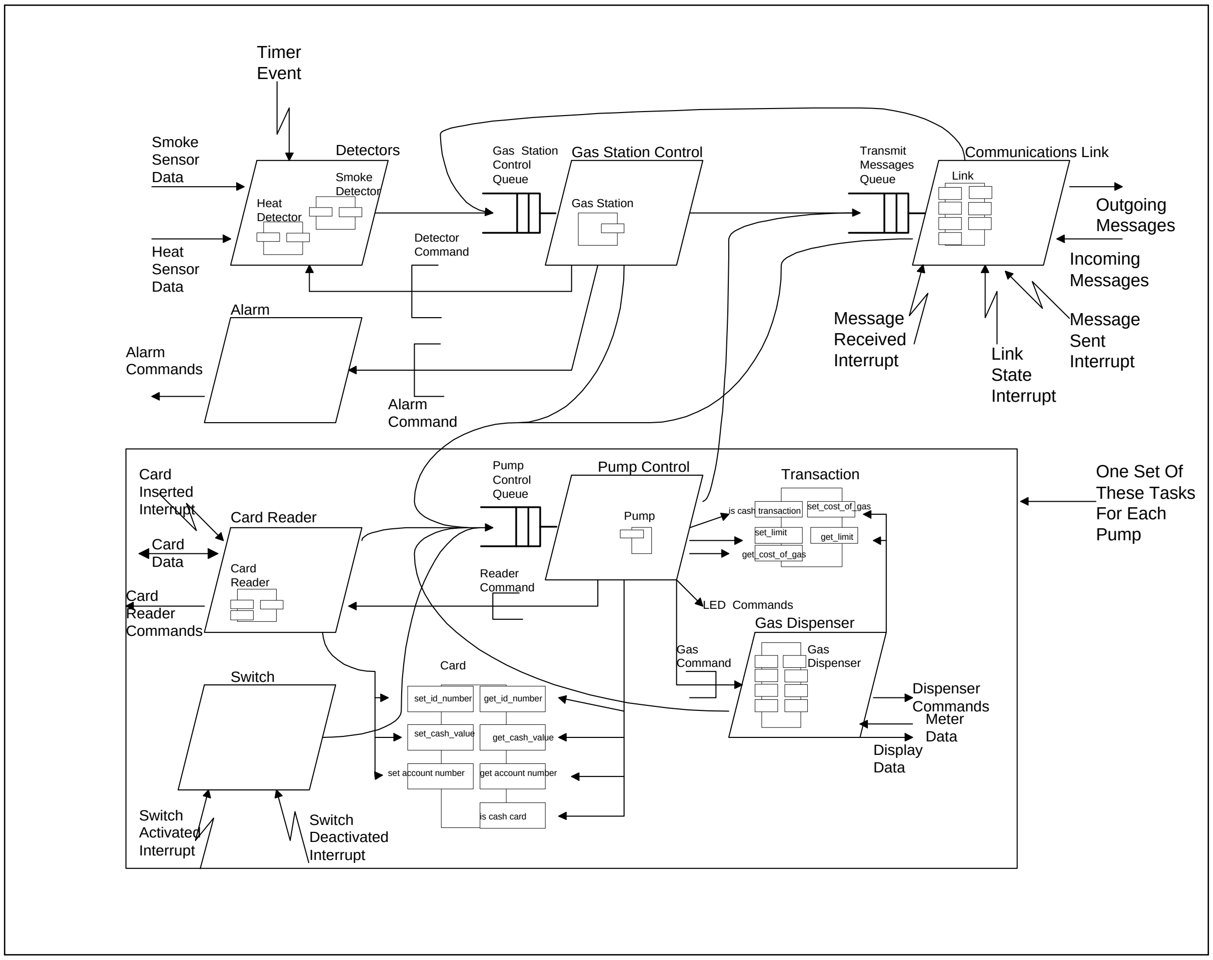


Figure 6. ADARTS System Architecture for AGMS from a COBRA Specification

Within the Detectors task, we included one IHM for each detector in the gas station. Each specific IHM depends on the type of detector, smoke or heat. We placed the gas station IHM, encapsulating the gas station control state transition diagram, inside the Gas Station Control task. Inside the Communications Link task, we placed a Link IHM that provides all operations needed to handle communications services. We placed the Pump IHM, encapsulating the Control Pump state transition diagram, inside the Pump Control task. The Gas Dispenser IHM, which hides the interface to the gas dispenser hardware, was placed into the Gas Dispenser task. The Card Reader IHM, which hides the details of the card reader hardware, we allocated to the Card Reader task. Since the Card and Transaction IHMs had already been placed between tasks, we had merely to add the operations supported by the IHMs and then to show which operations were invoked by which tasks.

The interested reader is advised to consult the TBS for each task and the specification for each IHM to gain an understanding of the operation of the system. Enough detail is provided so that a programmer can begin implementation of tasks and modules. For those readers who simply desire a more comprehensive overview of our ADARTS design for the AGMS we recommend the task and system architecture overviews beginning on pages E-2 and E-40, respectively. Now, we move on to discuss our OODARTS design for the AGMS.

B. OODARTS -- Altering the Recipe

Although ADARTS makes a wonderful recipe for obtaining designs from analyses, the method was developed absent additional ingredients that come with a full, object-oriented model. The ingredients available from an object-oriented analysis technique, such as OMT, tend to be fewer, but richer than those available from RTSA and COBRA. RTSA provides functions as ingredients for the design. These functions can be viewed as individual herbs and spices. COBRA adds objects in addition to functions. Such objects can be viewed as prepackaged combinations of herbs, spices, and other seasonings. OMT ingredients for the design are all prepackaged, i.e., OMT provides only objects. In addition, OMT ingredients can be altered through special handling that takes advantage of the advanced properties of such ingredients. OODARTS provides a recipe that employs some of the special handling available with OMT ingredients. And OODARTS builds on the existing recipe given by ADARTS.

ADARTS can be used to create a decent design from an OMT specification, but a number of concepts, such as inheritance and polymorphism, in the object-oriented paradigm cannot be exploited. Also, a strict object-oriented model requires that all units in the design be represented as objects, while ADARTS, relying on tasks and information hiding modules as its major building blocks, does not recognize the concept of an object. On the other hand, ADARTS provides the multiple threads of control needed in a concurrent design, while object-oriented approaches hand-wave about each object potentially executing under its own thread of control.

ADARTS provides criteria for structuring tasks from objects and functions, for structuring IHMs from objects and functions, and for allocating IHMs to tasks. When starting from an OMT specification, all functions have already been allocated to objects, and so, there is no need to allocate functions. And, since each object can potentially possess a thread of control, the main goal of an object-oriented design method should be to determine which objects have an independent thread of control (i.e., are active objects) and which do not (i.e., are passive objects). Then, for each active object, a thread of control must be specified.

On the basis of these observations, we devised an Object-Oriented Design Approach for concurrent Real-Time Systems (OODARTS), from the foundation established by ADARTS. While a future refinement of OODARTS might show a greater divergence from ADARTS, our initial development of OODARTS can be traced easily from ADARTS. We will explain our approach before presenting our OODARTS design for the AGMS.

The first phase in OODARTS requires that an active object (AO) architecture be developed. An AO possesses a independent thread of control; thus, an AO is analogous to an ADARTS task, except that an AO is an object. We require that each AO have at least two operations: 1) Create and 2) execute. The Create operation establishes the connections to other AOs in the architecture, sets up initial attribute values, and encapsulates the creation of any passive objects used by the AO. An AO can have any additional operations required. We can envision an AO class that has default Create and execute operations, that is inherited by each AO in the design, and that is refined to fit the behavior required for the type of object that is being made active.

To facilitate communication among AOs, we define an number of other concepts. An ACTIVE QUEUE provides a receptacle where loosely-coupled messages passed to an AO can be placed until the AO requests them. This is analogous to an ADARTS message queue. We can also envision an ACTIVE PRIORITY QUEUE, analogous to an ADARTS priority message queue.

Messages can be sent by AOs. Messages sent to other AOs are analogous to ADARTS tightly-coupled messages. Messages sent to ACTIVE QUEUES are analogous to ADARTS loosely-couple messages. Messages sent to ACTIVE PRIORITY QUEUES are analogous to ADARTS loosely-coupled messages sent with a specific priority. We define an operation **send** MESSAGE **to** DESTINATION [**with** REPLY]. Here MESSAGE can be viewed as a system object that can be inherited. The **send** operation causes different behavior depending on a number of factors. If the DESTINATION specifies an ACTIVE QUEUE, then the MESSAGE is placed at the end of the identified queue and the sender can continue operation. If the DESTINATION specifies an ACTIVE PRIORITY QUEUE, then the MESSAGE is placed at the end of the indicated priority slot for the identified queue and the sender can continue. Invalid or omitted priority designations result in the MESSAGE being placed in the lowest priority slot. If the DESTINATION specifies an ACTIVE OBJECT, then the MESSAGE is placed on a queue for that object and the sender is suspended until the DESTINATION accepts the MESSAGE. If the DESTINATION specifies an ACTIVE OBJECT and also includes an optional **with** REPLY, then, after placing the MESSAGE on a queue for the DESTINATION, the sender is suspended until a REPLY arrives from the DESTINATION.

An AO receiving messages has a number of mechanisms to use. The **Wait for** MESSAGE **from** SOURCE primitive causes the receiving AO to suspend until a message arrives from the specified source AO. The **Await** primitive causes an AO to suspend until any message arrives for the AO. The **Wait for** MESSAGE **in** ACTIVE QUEUE primitive causes an AO to suspend until a message arrives in the named queue. External events are handled by having an AO create appropriate interrupt vectors, and then by assigning interrupt handlers to those vectors. These mechanisms model the semantics available with ADARTS, but the semantics are presented in an object-oriented fashion.

In addition to defining an AO architecture, OODARTS requires a behavior specification, analogous to the task behavior specifications in ADARTS, for each AO. These AO behavior specifications: 1) name the AO, 2) specify the inputs and outputs of the AO in terms of events, messages, and data, 3) identify any passive object classes referenced by the AO, 4) indicate the criteria used for determining that the object requires an independent thread of control and identify any passive object classes included within the AO, 5) describe how the AO is activated and with what priority it executes, 6) detail the Create and execute operations, as well as any other necessary operations, for the AO, and 7) define any errors that the AO detects or avoids.

Beyond the AO architecture and behavior specifications, OODARTS requires class specifications for each passive object class in the design. Such object class specifications replace the IHM specifications in the ADARTS method. An OODARTS object class specification: 1) names the object class, 2) explains what is encapsulated by the object class, 3) classifies the object class using criteria from ADARTS, 4) describes any assumptions made about the object class, 5) outlines any changes that are planned for the object class, 6) defines the attributes encapsulated within the object class, and 7) specifies the operations provided by the object class. As with ADARTS, passive object classes are allocated within the AO architecture to produce a system architecture for the design.

Using this OODARTS method, we derived a design for the AGMS from our earlier OMT analysis. An overview of the Active Object Architecture is presented on pages F-1 through F-5. The architecture is shown below as Figure 7.

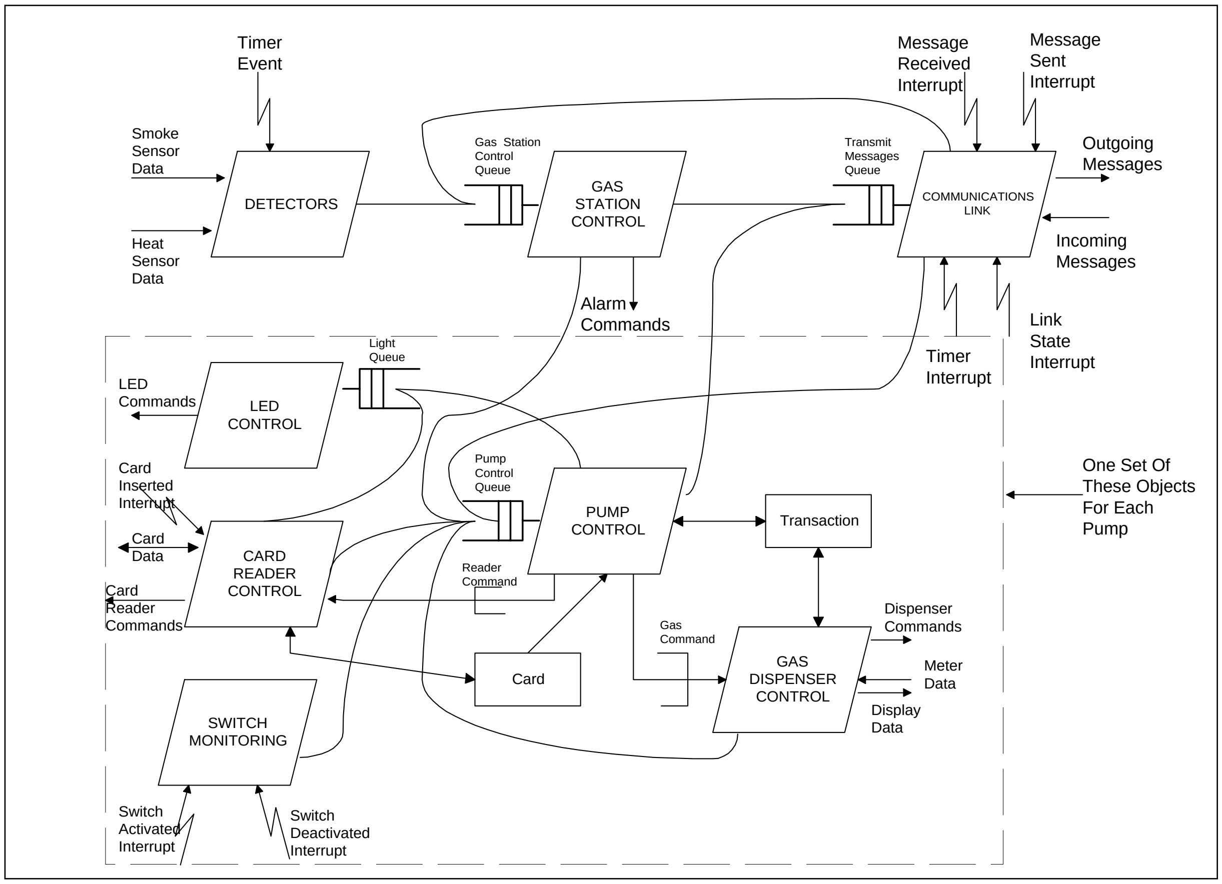


Figure 7. OODARTS Active Object Architecture for the AGMS from an OMT Specification

In Figure 7, each active object is shown as a parallelogram, while each passive object that is used by multiple active objects is shown as a rectangle. The other notation is also adopted from ADARTS. The architecture we achieved from the OMT specification, using OODARTS, was essentially the same as that we devised from the COBRA specification, using ADARTS. The astute reader will notice some differences between the two architectures; however, these differences stem from different interpretations of some requirements, not from any difference in the methods. For example, the ADARTS design included an Alarm task, while the OODARTS design does not define an Alarm AO. This results because during the COBRA analysis we assumed that the alarm device needed constant pulsing to sound it, while in the OMT analysis we assumed that the alarm device was a two-state device that would remain in whatever state it was set. The other difference involves LED control. In the COBRA analysis we assumed that if the customer inserted an unrecognized card, then the card would simply be ejected, while in the OMT analysis, we assumed that the "Cannot Process Card Light" would be lit before the card was ejected. The first assumption means that the LED control can be sequentially included with the Pump Control task, while the second assumption means that two AOs, Card Reader and Pump Control, share access to the LEDs, and, thus, an LED AO is required as a resource monitor. Had the same assumptions been made during both the COBRA and OMT analyses, then the resulting ADARTS and OODARTS architectures would be identical.

A behavior specification is given for each AO shown in Figure 7. These specifications can be found on pages F-6 through F-31. The interested reader is advised to scan the AO architecture overview (beginning on F-2), and then to investigate the behavior specifications for any AOs that appear interesting. Next, we recommend that the reader examine the system architecture on pages F-52 to F-54, and then to skim the object class specifications for any passive objects that seem pertinent. In the remainder of this section, we describe how we applied the OODARTS method to arrive at the design in Appendix F, from the OMT specification in Appendix D.

Our first step was to examine the object, dynamic, and functional models of our OMT specification to identify active objects. We applied as many of the ADARTS task structuring criteria as we could. We envisioned an AO for each object in the OMT specification that possessed a state chart. This identified the Detectors, the Gas Station, the Communications Link, the Switch, and the Pump as candidate AOs. Since we envisioned that the Detectors would be polled on a regular basis, we decided to encapsulate all of the Detectors into a single AO (i.e., Detectors on page F-3). We found by examining the OCDs in the functional model that the Gas Station, Communications Link, and Pump objects received input events from multiple sources, so we retained these candidates as AOs: Gas Station Control, Communications Link, and Pump Control. Since the Switch object interacted with external events, we encapsulated the Switch object within a Switch Monitoring AO.

We then examined the remaining objects from the OMT object model to identify any additional AOs. The Card Reader object is activated by external events, so we encapsulated the Card Reader within a Card Reader Control AO. We also found that the LED objects served multiple AOs, and therefore we encapsulated the LED objects within an LED Control AO to monitor the LEDs as shared resources. We found that the Gas Dispenser object could be activated (to run under its own control and complete operations under internally recognized conditions) and deactivated by the Pump Control AO. For these reasons, we encapsulated the Gas Dispenser inside a Gas Dispenser Control AO.

We viewed the remaining objects, the various cards and transactions, as data encapsulation objects that execute sequentially under control of the AOs. The card objects are used by the Card Reader Control and Pump Control AOs, and the transaction objects are used by the Pump Control and Gas Dispenser AOs. Although we could have moved the card and transaction objects inside the Pump Control AO, we chose to design them as passive objects, external to all AOs, and shared between AOs.

The context diagram from our OMT functional model leads to a natural allocation of external events and data to particular AOs, and so our next design decisions focused on internal communications between AOs. Here, again, the dynamic model and the OCDs from the functional model helped us make choices. Because the Gas Station Control AO received inputs asynchronously from the Detectors and Communications Link AOs, we defined an active queue (AQ), called the Gas Station Control Queue. Similar reasoning, differing only in the specifics, led us to define the Transmit Messages Queue, the Light Queue, and the Pump Control Queue. The Card Reader Control AO serves only the Pump Control AO by awaiting Reader Commands to arrive and then executing them; thus, we chose tightly-coupled communications between the two AOs. The Gas Dispenser Control AO waits for start and stop Gas Commands from the Pump Control AO and then executes them; thus, again, we chose tightly-coupled communications between the two AOs. The details of the messages flowing into the various AQs and between the various AOs can be mapped readily from the OMT specification. We so mapped them, and then we documented them as message exchanges within each AO behavior specification.

Next, we turned our attention to the passive object classes. The allocation of passive objects among the AOs was, for the most part, decided previously, when we created the AO architecture. This follows from the fact that both OMT and OODARTS use the object as the only unit. When we created the AO architecture, we were allocating OMT objects (i.e., passive objects). The objects were allocated either to an AO, or between several AOs. Any objects that we did not allocate while developing the AO architecture, we allocated now. For the AGMS design, we allocated the Alarm object to the Gas Station Control AO, because the Alarm object coheres sequentially to the Gas Station object.

Since the passive object classes were identified during the analysis, we need only specify, during the OODARTS design process, each object class. Here, again, most of the work was accomplished during the analysis. We used the complete object model from the OMT analysis, amplified by the object dictionary and the object function descriptions, to create the OODARTS object class specifications shown on pages F-32 through F-51. After documenting the object class specifications, we updated the AO behavior specifications to reflect particular details of the operations defined for the passive objects used by each AO.

Next, we updated the AO architecture diagram to reflect our allocation of passive object classes among AOs. This action yielded a system architecture diagram for the AGMS, as shown in Figure 8 (and also on page F-54). Here, we identified the two types of card object (Credit Card, as distinct from Cash Card) and the two types of transaction object (Credit Transaction, as distinct from Cash Transaction) that are shared among AOs. For each of these shared object classes, we specified the operations provided by the object class, as well as illustrated which operations are invoked by which AOs. Where a passive object sends a message to an AO or AQ, we depicted that as well. As a final step in our OODARTS design for the AGMS, we wrote a brief overview of the system architecture (see page F-53).

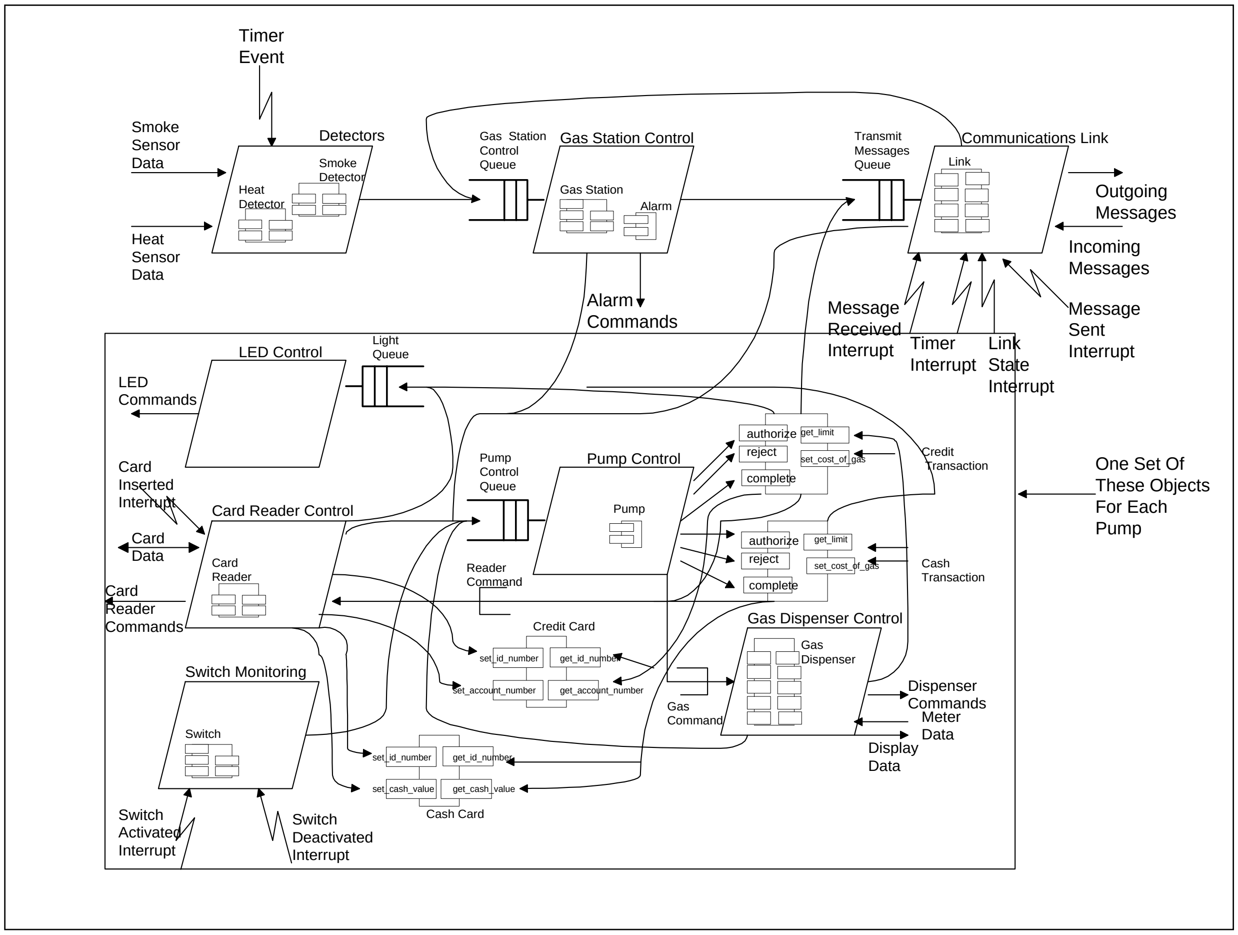


Figure 8. OODARTS System Architecture for the AGMS from an OMT Specification

Although our system architecture keeps AOs separate from the passive objects that they encapsulate, a more integrated approach is possible. Once we identified active objects from our OMT specification, we could have added the necessary Create and execute operations to those objects, transforming them from their passive, OMT form to an active, OODARTS form. This approach would work well for the Gas Station, Pump, Link, Card Reader, Gas Dispenser, and Switch objects. (Our design would still benefit from encapsulating multiple instants of Detector objects into a single AO.) We chose to maintain a separation between the AOs defined in our OODARTS design and the passive objects identified in our OMT specification. This choice should enable the reader to see easily how OODARTS was developed from the ADARTS method.

V. Evaluation Of Results

In this paper we stalked an automated gas station management problem with three analysis methods and two design approaches. We now approach the weighing station at the game warden’s shelter. Here we must assess the results of our safari. The rules of the preserve in which we have hunted allow us to assess our results in our own words, but the game warden might come along for a spot check. We imagine that the reader will take on the role of game warden. We begin with an assessment of the analysis techniques we used.

RTSA proved simple to use. The concept of data flow diagrams is familiar to most analysts, and even someone without analysis experience can catch on quickly. The concept of finite state automata is mature. RTSA incorporates finite state automata into structured analysis using a natural, easily comprehended model, the control transform. The specifications produced from an RTSA analysis are easy to understand and easy to maintain. On the other hand, RTSA appears applicable only to small problems. RTSA analysis leads to a large number of small functions activated by control transforms. While the decomposition techniques supported by data flow diagrams allow for chunking and aggregating functions, most resulting RTSA specifications for large problems would be difficult to comprehend. Also, RTSA specifications point to specific design decisions; thus, RTSA is not solution independent. But, then, none of the analysis techniques we used are solution independent; RTSA seemed more solution independent than either COBRA or OMT.

COBRA appeared more useful for analyzing large, real-time problems. The subsystem decomposition guidelines of COBRA enable a problem to be divided into fairly independent subsystems. Each subsystem can then be analyzed separately, studied separately, and understood separately. COBRA builds on the concepts and notation of RTSA. This relationship between COBRA and RTSA should enable experienced RTSA analysts to begin using COBRA without much training. COBRA also provides a number of guidelines to help the inexperienced analyst. These include: 1) guidelines for developing the COBRA environmental model, 2) guidelines for decomposing a problem into subsystems (or sub-problems), 3) criteria for determining objects and functions, and 4) procedures for performing behavioral scenario analysis. The COBRA guidelines help to ensure that COBRA specifications are complete, consistent, and repeatable.

Despite the improvements achieved by COBRA, we noticed some drawbacks to the technique. For example, COBRA specifications contain redundant information. The behavioral scenario analysis that develops the state transition diagrams and data/control flow diagrams leads to two specifications for each state transition diagram and to many instances for some of the data and control transforms. This provides a means to check the consistency of the analysis, but also increases the difficulty of maintaining the specification. Another difficulty we experienced while using COBRA involved functions and objects. Since COBRA allows the specification of both functions and objects, we often had to decide when to specify a function and when to specify an object. Even with the criteria included in COBRA, we often found ourselves unsure when to specify functions and when to use objects. These decisions are important because COBRA is not solution independent. In fact, COBRA leads toward an object-based implementation, perhaps imagining a language like Ada for the ultimate implementation. For this reason, we found COBRA to be more solution-oriented than RTSA.

Where COBRA and RTSA are related analysis techniques, OMT belongs to a different family. We found the greatest strength of OMT to be the unifying concept of the object model. The early stages of OMT required us to mine the problem domain for objects, attributes, and relationships. The object model that resulted from this analysis provides the unifying framework for the remaining stages of OMT analysis. OMT provides good guidance for identifying objects, attributes, and relationships, although at first we had difficulty crafting the relationships properly. Since OMT relies on an object model, we could take full advantage of object-oriented concepts such as inheritance and polymorphism. Another positive result from OMT is that functions are allocated to objects during the analysis, so further consideration of design issues deals only with objects, not functions. Another strength of OMT appears in the dynamic model. Because OMT adopts state charts (in the place of state transition diagrams), we could take advantage of a number of notational and semantic conveniences associated with state charts. We also found that using OMT led us to an increased use of dynamic models. For example, using RTSA and COBRA we identified two state transition diagrams, while with OMT we created five state charts. We even wondered if we might not have been wise to create additional state charts for the Gas Dispenser and the Card Reader.

While OMT provided some useful improvements over COBRA and RTSA, we found OMT to contain a number of flaws. Probably the most serious flaw was the lack of integration between the functional model and the object and dynamic models. We could not readily model the automated gas station problem as a data flow diagram without the use of control transforms. Yet, control transforms are not permitted in the OMT functional model. To solve this shortcoming, we developed an alternate approach using object communication diagrams and object function diagrams. Our alternate approach allowed us to identify functions and to allocate functions to objects in our object model. We cannot expect every analyst to develop their own solution for functional modeling in OMT, but the solution given in the Rumbaugh book will not always lead to a successful result. Another problem we believe exists with OMT is lack of scaling. Object models can become quite complex. OMT provides notational devices to divide an object model among multiple sheets of paper; however, the fundamental complexity of the model is not improved by such notational devices. Because of this flaw, we believe that OMT can only be applied to problems of moderate size. Were OMT to introduce a subsystem concept and then to give guidelines for decomposing a problem into subsystems (as COBRA does), we believe that OMT would scale to larger problems. A third fault we found with OMT is lack of a rigorous approach to develop state charts. We were always wondering where we should define a state chart and where we should not. We also could have benefited from a precise method for developing each state chart. (Perhaps something similar to the COBRA behavioral scenario analysis method could be added to OMT.) A final shortcoming we noted with OMT is that many, if not most, design decisions are made during the analysis. In fact, all road signs from an OMT specification point to an object-oriented implementation. We were not dismayed by this outcome because we intended to develop an object-oriented design for the automated gas station problem. We advise those analysts not heading toward an object-oriented design to avoid OMT during the analysis.

After we developed our three analysis specifications for the automated gas station problem, we produced two designs. One design used ADARTS and the other used OODARTS (a method we devised from ADARTS). Here we consider the results from applying ADARTS and OODARTS. Remember that we developed the ADARTS design from the COBRA specification and we produced the OODARTS design from the OMT specification. We begin with ADARTS.

ADARTS provides a reliable recipe for deriving concurrent designs from RTSA and COBRA specifications. The resulting design is Ada-based. Since many Ada-based environments exist on a variety of processors, ADARTS designs can be readily implemented in Ada. Even in the absence of an Ada compiler and run-time environment, ADARTS designs can be mapped onto most real-time executives or operating systems. One possible shortcoming with ADARTS involves the limited use of information hiding modules (IHMs). ADARTS encourages IHMs to be restricted to encapsulation of data stores, device interfaces, and state transition diagrams. Although ADARTS permits a greater use of IHMs, many practical applications of ADARTS encapsulate objects into IHMs that are allocated inside tasks and relegate IHMs shared between tasks to a more minor role of encapsulating data stores. Another possible shortcoming of ADARTS is the requirement that the designer allocate both objects and functions to tasks. This shortcoming results from the nature of the RTSA and COBRA analysis techniques, coupled with the fact that ADARTS is Ada-based rather than object-oriented.

We developed OODARTS as an object-oriented extension to ADARTS. OODARTS assumes an object-oriented support environment, rather than an Ada-based support environment. For this reason, we had to define a support environment for OODARTS. The environment we defined combined ADARTS semantics with object-oriented concepts. OODARTS, then, provides for full, object-oriented designs that lead naturally into an object-oriented programming solution. Of course, an underlying object-oriented programming model for concurrent systems must be implemented. No such model is widely accepted nor implemented. Still, we outlined a means for mapping a concurrent, object-oriented model onto most real-time executives or operating systems. Our method requires that an object-oriented layer be implemented (using an object-oriented language) above the real-time execute. Without a run-time environment to support OODARTS some of the object-oriented features of the design must be discarded. If such features are discarded, then ADARTS can be used to generate a design from an OMT specification. Of course, some of the traceability to the original analysis will be lost. OODARTS yields reasonable, concurrent, object-oriented designs from OMT specifications. In fact, in our exercise with the automated gas station manager, the design resulting from COBRA/ADARTS was identical fundamentally to the design resulting from OMT/OODARTS. In the former case, the design can be easily taken to an Ada implementation. In the latter case, the design can be directly taken to an object-oriented programming implementation, provided that an underlying OODARTS run-time environment exists.

VI. Conclusions

Two trends in the computer industry appear at odds. One trend is the growing need for concurrent and distributed, real-time systems to control an increasingly digital world. The other trend is the expanding popularity of object-oriented programming (OOP). OOP traditionally has ignored many practical aspects of system requirements analysis and design. In fact, only in the last few years have methods for object-oriented analysis and design (OOA and OOD) emerged. These OOA and OOD methods are intended to provide a smooth path from problem analysis to programming implementation. Unfortunately, many of the existing object-oriented methods ignore problems associated with concurrency and distribution and with other requirements of real-time systems.

In the foregoing paper, we have shown that OOA/OOD/OOP can be used to develop effective analyses, designs, and implementations for concurrent, real-time problems. We used an automated gas station management problem as our example. We showed how such a problem could be analyzed with some well-known techniques such as RTSA and COBRA. We also showed how an object-oriented method, OMT, could be used to analyze the problem. We then showed how ADARTS could be used to produce a concurrent design from a COBRA specification. We explained how we derived an object-oriented design approach (OODARTS) from ADARTS and we applied OODARTS to produce a concurrent design from our OMT specification. The two resulting concurrent designs were identical fundamentally. From this exercise, we concluded that OOA/OOD/OOP can lead to effective concurrent designs for solutions to real-time problems. We further concluded that the key to making a smooth transition from OOD to OOP for a concurrent design is the existence of an underlying object-oriented, run-time model (implemented in an object-oriented language) for managing multiple threads of control and inter-object communication and synchronization. Such a model would be analogous to the Ada run-time environment. Unfortunately, no such object-oriented, run-time model is accepted widely. Without such a model, moving from a concurrent OOD to an OOP implementation will remain a difficult art.

VII. References

[1] Philip A. Laplante, Real-Time Systems Design And Analysis An Engineers Handbook, IEEE Computer Society Press: Los Alamitos, CA, 1993, 339 pages.

[2] P.T. Ward and S. J. Mellor, Structured Development for Real-Time Systems, Volumes I, II, and III, Yourdon Press, New York, 1985 - 1986.

[3] Hassan Gomaa, Course Notes On Software Design Methods, Parts I and II, George Mason University, Fall Semester 1992, Chapter 5: *Design Approach for Real-Time Systems*.

[4] David A. Watt, Brian A. Wichmann, and William Findlay, ADA [sic] Language and Methodology, Prentice-Hall: Englewood Cliffs, NJ, 1987, 518 pages.

[5] Hassan Gomaa, Course Notes On Software Design Methods, Parts III and IV, George Mason University, Fall Semester 1992, Chapter 14: *Analysis and Modeling for Concurrent and Real-Time Systems*.

[6] Hassan Gomaa, Course Notes On Software Design Methods, Parts III and IV, George Mason University, Fall Semester 1992, Part III: *A Design Approach for concurrent and Real-Time Systems (ADARTS)* and Part IV: *ADARTS Case Studies*.

[7] Bjarne Stroustrup, The C++ Programming Language, Second Edition, Addison-Wesley: Reading, Mass., 1991 (reprinted with corrections in 1992), 669 pages.

[8] Bertrand Meyer, Eiffel: The Language, Prentice-Hall International: Hemel Hempstead, United Kingdom, 1992, 594 pages.

[9] Bertrand Meyer, Object-oriented Software Construction, Prentice-Hall International: Hemel Hempstead, United Kingdom, 1988, 534 pages.

[10] Grady Booch, Object-Oriented Design With Applications, Benjamin/Cummings, Redwood
City, CA, 1991, 580 pages.

[11] Sally Shlaer and Stephen J. Mellor, Object Lifecycles, Modeling the World in States, Yourdon Press: Englewood Cliffs, NJ, 1992, 251 pages.

[12] James Rumbaugh, et al., Object-Oriented Modeling and Design, Prentice-Hall: Englewood
Cliffs, NJ, 1991, 500 pages.

[13] Peter Coad and Edward Yourdon, Object-Oriented Analysis, Second Edition, Yourdon

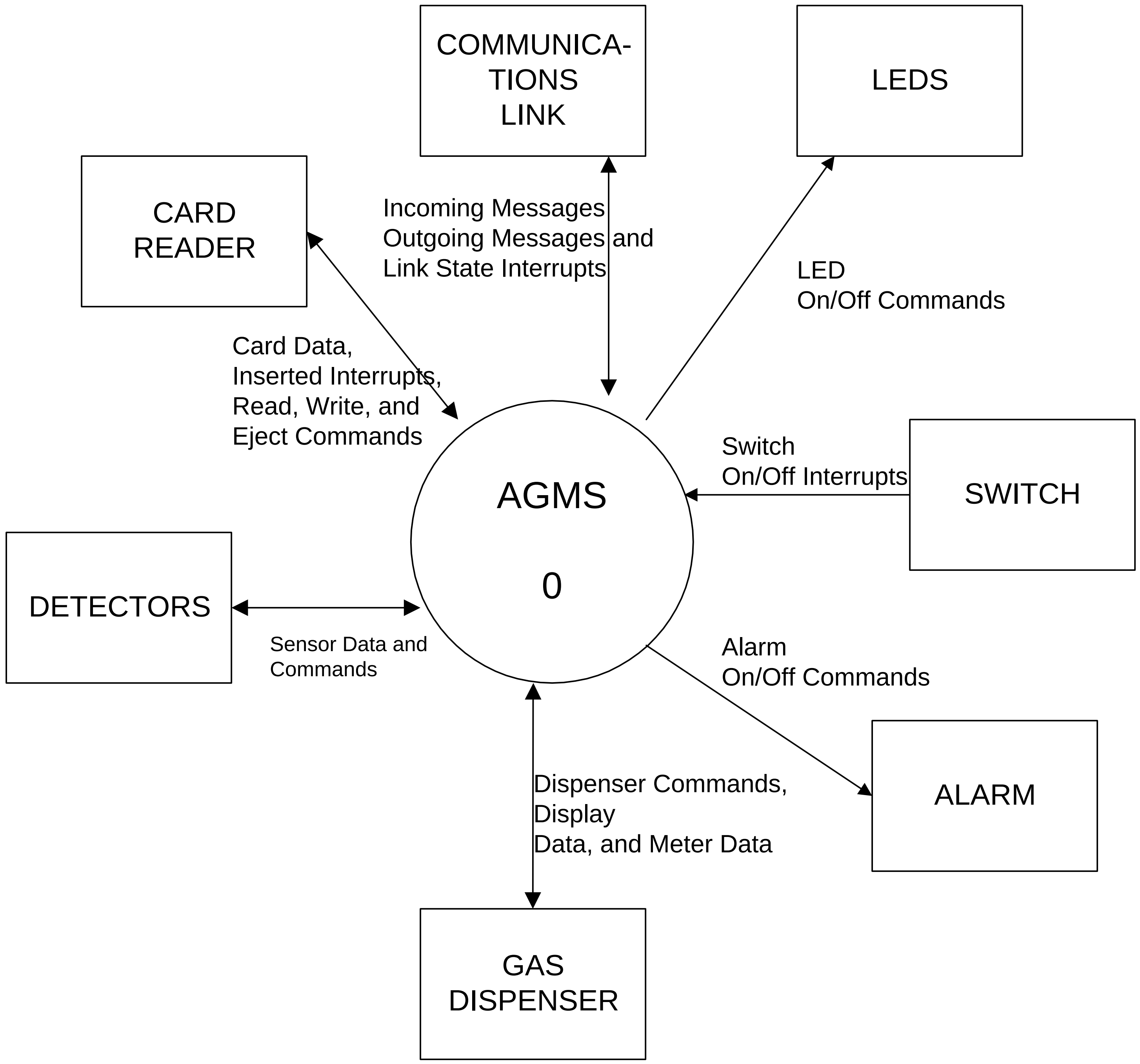
Press: Englewood Cliffs, NJ, 1991, 233 pages.

[14] John Palmer, *Object-oriented analysis and polymorphism: Gross concept neglect*, Object Magazine, March-April, 1993, pp. 34-36.

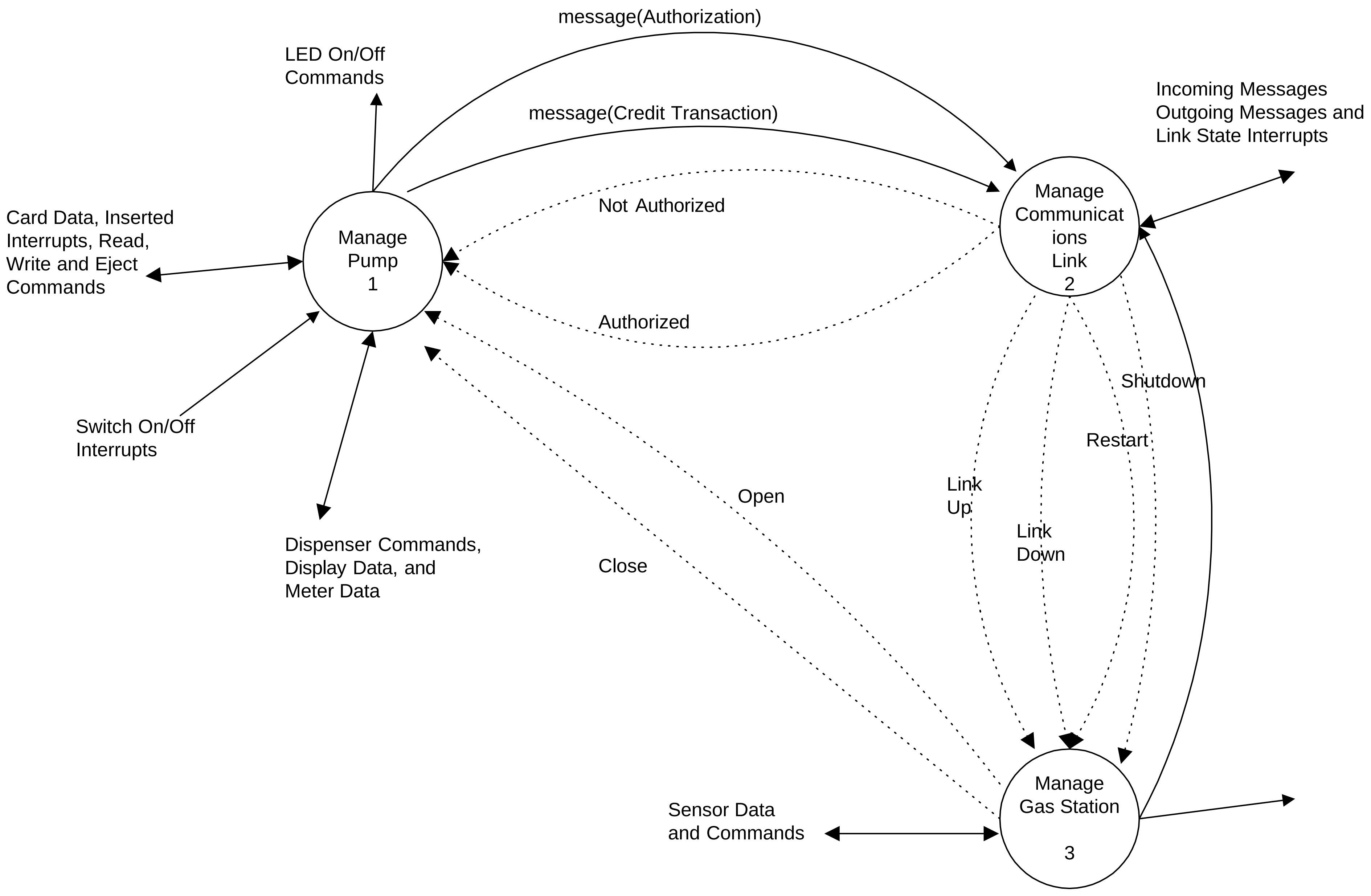
APPENDIX B. AUTOMATED GAS STATION MANAGER RTSA SPECIFICATION

AUTOMATED GAS STATION MANAGER

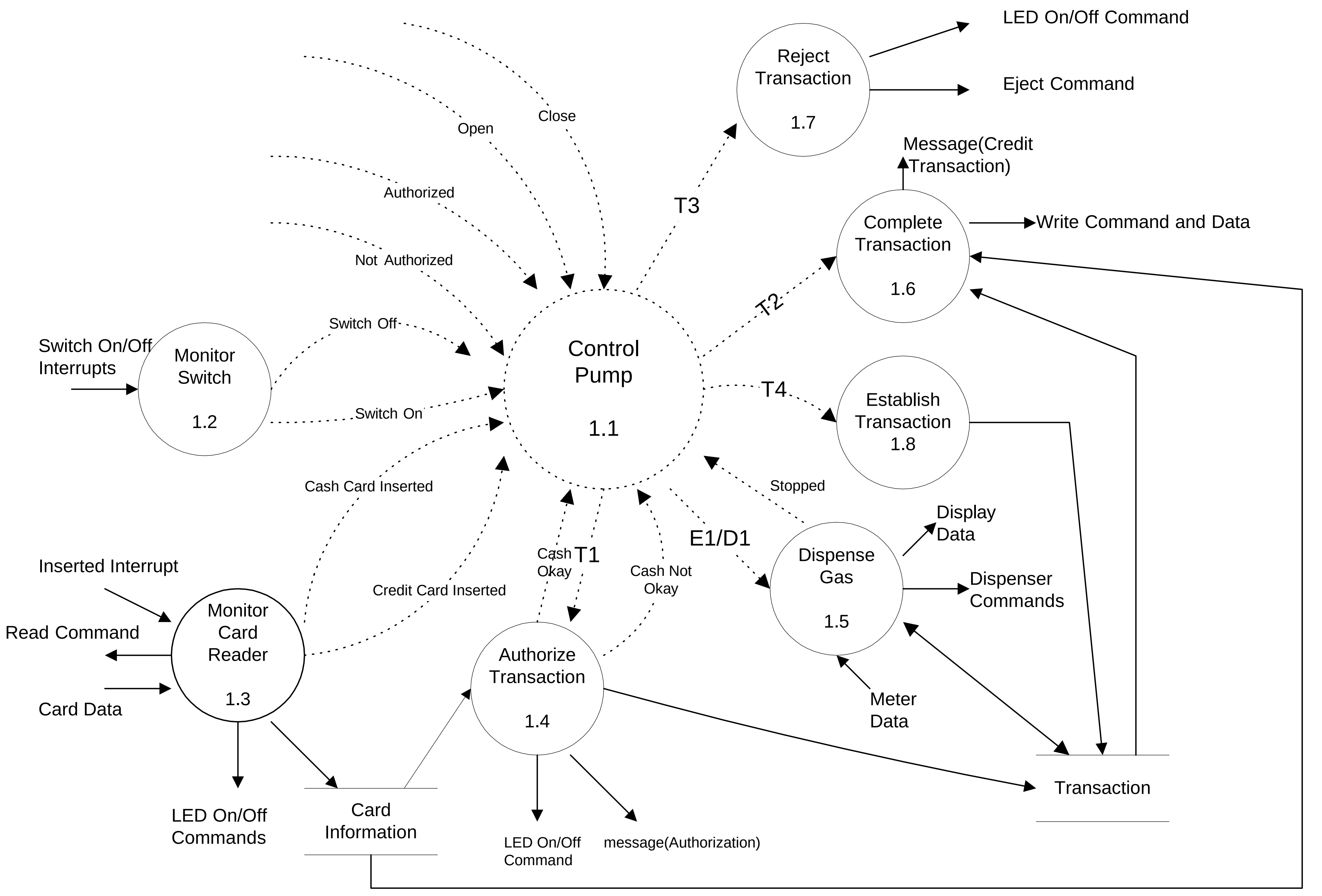
RTSA DATA/CONTROL FLOW DIARGRAMS



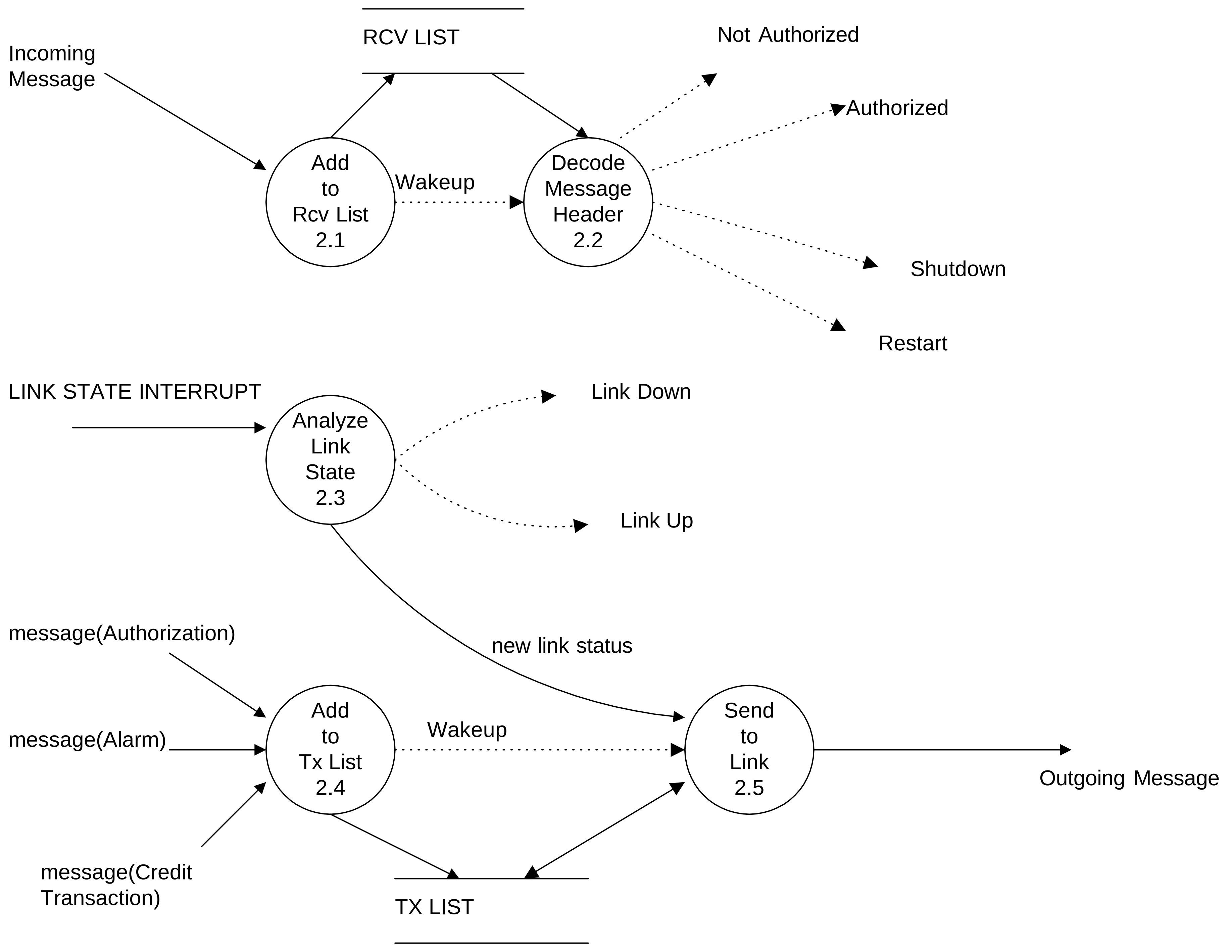
B-2 AGMS CONTEXT DIAGRAM


message(Authorization)
LED On/Off Commands
message(Credit Transaction)
Incoming Messages Outgoing Messages and Link State Interrupts
Manage Communications Link 2
Not Authorized
Card Data, Inserted Interrupts, Read, Write and Eject Commands
Manage Pump 1
Authorized
Shutdown
Switch On/Off Interrupts
Restart
Link Up
Open
Link Down
Dispenser Commands, Display Data, and Meter Data
Close
Manage Gas Station 3
Sensor Data and Commands

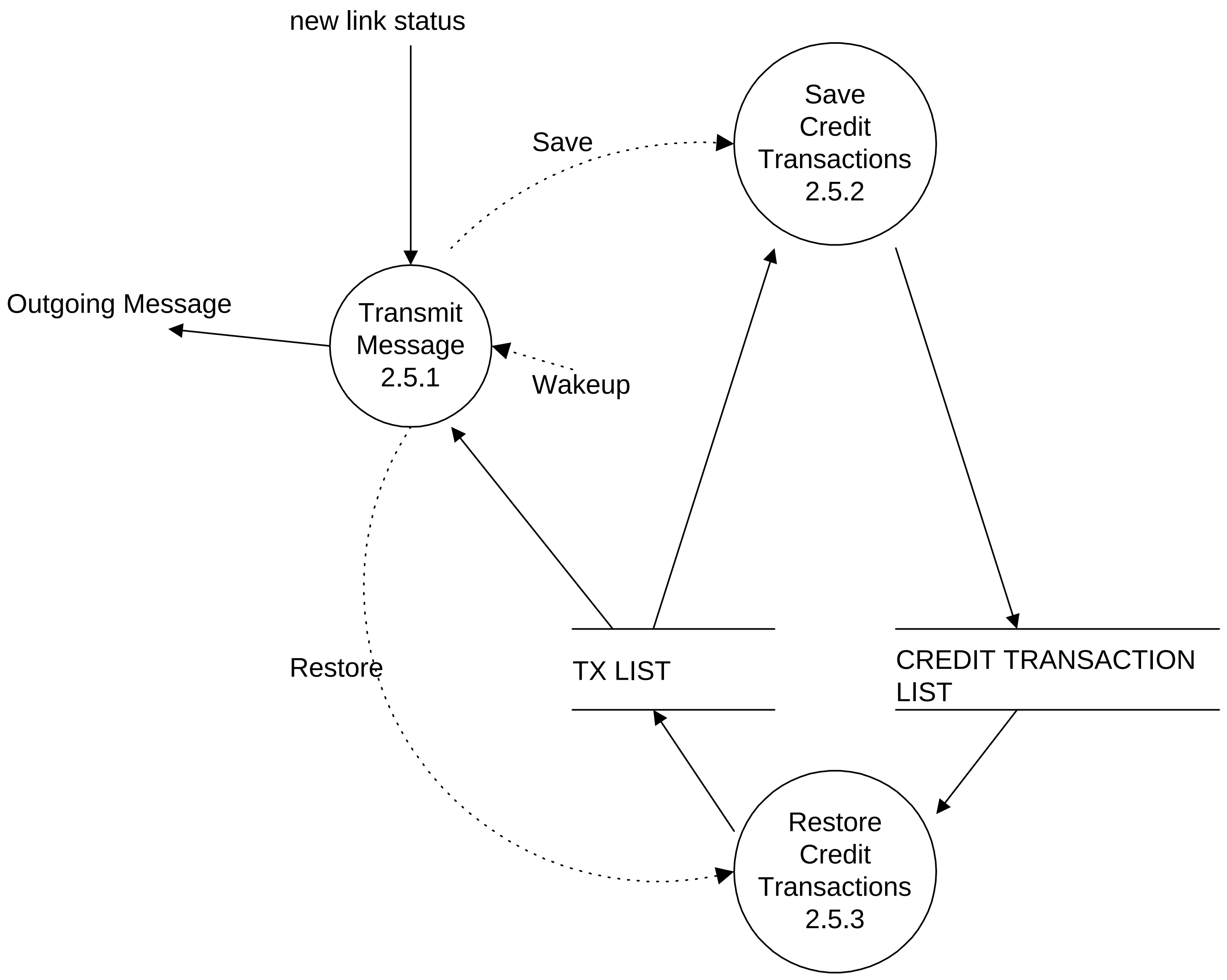
B-3 AGMS DECOMPOSITION INTO SUBSYSTEMS



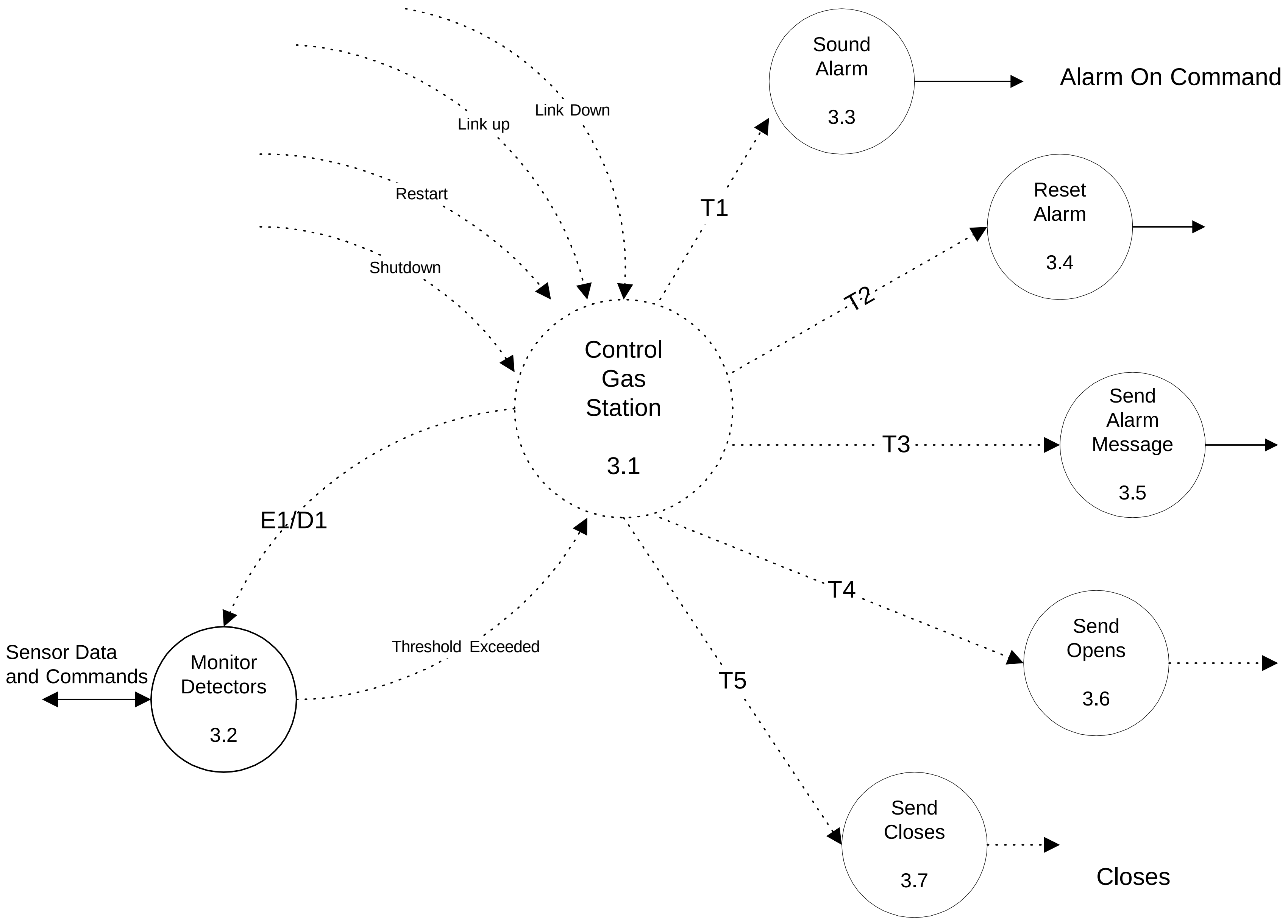
B-4 1 MANAGE PUMP CONTROL/DATA FLOW DIAGRAM



B-5 2 MANAGE COMMUNICATIONS LINK DATA FLOW DIAGRAM



B-6   2.5 Send to Link DATA FLOW DIAGRAM



B-7 3 MANAGE GAS STATION CONTROL/DATA FLOW DIAGRAM

AGMS DATA DICTIONARY

Alarm Commands

0 is TURN OFF
1 is TURN ON

Card Data : RECORD

-- Read from an inserted Card

1. identification number : ASCII STRING	-- Determines second field
2. account number : ASCII STRING	-- If Credit Card
3. cash value : ASCII STRING	-- If Cash Card

Card Information: RECORD

--Holds the data read from a Card inserted into the Card Reader

1. identification number: NUMERIC	
2. account number: NUMERIC	-- Valid for Credit Card Only
3. cash value: INTEGER	-- Valid for Cash Card Only

Card Read Commands

0 is Eject
1 is Read (Input Data is a Card Data Record)
2 is Write (Output Data is a Card Data Record)

CREDIT TRANSACTION LIST: FILE

--A File containing saved Credit Transaction Messages

Dispenser Commands

0 is STOP GAS	
1 is START GAS	
2 is READ METER	(Input data is Meter Data)
3 is WRITE DISPLAY	(Output data is Display Data)

Display Data: RECORD

--Contains two values to be displayed

1. display gallons: FIXED POINT (Gallons and Tenths and Hundreths)
2. display cost: FIXED POINT (Dollars and Cents)

Message: RECORD

--The Data Structure that holds a message

1. To: ADDRESS (Station or Remote Central Facility)
2. From: ADDRESS (Station or Remote Central Facility)
3. Type: BYTE
   - 0 is SHUTDOWN COMMAND
   - 1 is RESTART COMMAND
   - 2 is Credit Transaction
   - 3 is Alarm
   - 4 is Authorization request
   - 5 is Authorization reply
4. Optional additional parameters determined by Type.
   - For Credit Transaction:
     - Pump ID: INTEGER
     - account number: NUMERIC
     - cost of gas: INTEGER (in cents)
   - For Authorization Request
     - Pump ID: INTEGER
     - account number: NUMERIC
   - For Authorization Reply
     - Pump ID: INTEGER
     - response: BOOLEAN
       - 0 is Not Authorized
       - 1 is Authorized

Meter Data: REAL

-- Data read from the Gas Dispenser's meter

new link status: INTEGER

-- The new status of the communications link. Values can be 0, no change, 1, link up, or
-- -1, link down

RCV LIST: LINKED_LIST

-- This is the queue of messages received on the Communications Link, but not yet
-- processed.

Sensor Commands

0 is Enable	
1 is Disable	
2 is Read	(Input Data is Sensor Data)

Sensor Data: DEVICE DEPENDENT

-- The raw input from a sensor device. The form is dependent on the specific sensor.

Transaction : RECORD

1. STATION ID : INTEGER
2. PUMP ID : INTEGER
3. cost of gas : INTEGER
4. limit : INTEGER

TX LIST: LINKED_LIST

-- The queue of messages that are to be transmitted, but that have not yet been transmitted

AGSM Mini-Specifications

1.2 Monitor Switch

```
LOOP FOREVER
WAIT for ACTIVATE INTERRUPT OR DEACTIVATE INTERRUPT
IF ACTIVATE INTERRUPT
        THEN Send Switch ON to Pump STD
        ELSE  Send Switch OFF to Pump STD
ENDIF
END  FOREVER LOOP
```

1.3 Monitor Card Reader

```
LOOP FOREVER
        WAIT for CARD INSERTED INTERRUPT
        READ identification number and store in Card Information
        IF identification number is Cash Card
                THEN READ cash value,
                        store cash value in Card Information,
                        Send Cash Card Inserted Event to Pump STD
        ELSEIF identification number is Corporate Credit Card
                THEN READ account number,
                        store account number in Card Information,
                        Send Credit Card Inserted Event to Pump STD
        ELSE    CALL Reject Transaction
        ENDIF
END LOOP
```

1.4 Authorize Transaction

```
IF Card Information.identification number is for a Cash Card
        THEN
                IF cash value is not positive
                        THEN Output LED ON COMMAND to Cash Value Used LED
                                Send Cash Not Okay Event to Pump STD
                                Wait 10 Seconds
                                Output EJECT COMMAND to Card Reader
                                Output LED OFF COMMAND to Cash Value Used LED
                        ELSE    Create Transaction with STATION ID and PUMP ID
                                Set Transaction.limit to cash value
                                Send Cash Okay Event to Pump STD
                ENDIF
        ELSE    Authorization Request := BUILD (Remote Central Facility Address, Station ID,
                                Pump ID, Card Information.account number)
                CALL Add to Tx List(Authorization Request)
ENDIF
```

1.5 Dispense Gas

```
Send START COMMAND to Gas Dispenser
LOOP FOREVER UNTIL Disabled from Pump STD
        CALL Monitor Meter
        CALL Update Display
        If amount dispensed/100 * price per gallon
                >= cash limit THEN BREAK LOOP
END FOREVER LOOP
Send STOP COMMAND to Gas Dispenser
CALL Monitor Meter
CALL Update Display
Set Transaction.cost of gas to amount dispensed/100 * price per gallon
Send Stopped to Pump STD
RETURN
```

Monitor Meter

```
READ meter value
Convert to hundreths of gallons
Set amount dispensed to converted value
```

Update Display

```
display gallons := amount dispensed/100
display cost := display gallons * price per gallon
Output display gallons to GALLONS DISPLAY
Output display cost to COST DISPLAY
```

1.6 Complete Transaction

```
IF Card Information.identification number is for Credit Card
        THEN  Credit Transaction := BUILD(Remote Central Facility Address, Station ID, Pump
                                          ID, account number,
                                          amount dispensed/100*price per gallon)
              CALL Add to TX List (Credit Transaction)
        ELSE  new cash value := cash limit - (amount dispensed/100*price per gallon)
              Output WRITE COMMAND with Card Information.identification number and
                      new cash value to Cash Card
ENDIF
Output EJECT COMMAND to Card Reader
```

1.7 Reject Transaction

```
Output LED ON COMMAND to Cannot Process Card LED
Output EJECT COMMAND to Card Reader
Wait 10 Seconds
Output LED OFF COMMAND to Cannot Process Card LED
```

1.8 Establish Transaction

```
Create Transaction with STATION ID and PUMP ID
Set Transaction.limit to zero
```

2.1 Add to Rcv List

```
Accept message from Link
Add message to RCV LIST
CALL Decode Message Header
```

2.2 Decode Message Header

```
IF message is a RESTART COMMAND
        THEN generate a Restart event for the Gas Station STD
ELSEIF message is a SHUTDOWN COMMAND
        THEN generate a Shutdown event for the Gas Station STD
ELSEIF message is an authorization reply
        THEN IF message contains authorization
                        THEN generate an Authorized event for the appropriate
                                Pump STD
                        ELSE generate an Unauthorized event for the appropriate
                                Pump STD
                ENDIF
ENDIF
Remove message from the RCV LIST
```

2.3 Analyze Link State

```
IF LINK STATE is UP and was DOWN
   THEN CALL Transmit Message with new link state set to UP
ELSIF LINK STATE is DOWN and was UP
    THEN CALL Transmit Message with new link state set to DOWN
ELSE CALL Transmit Message with new link state set to NO CHANGE
```

2.4 Add to Tx List(message)

```
Add message to TX LIST
CALL Transmit Message(NO CHANGE)
```

2.5.1 Transmit Message(new link state)

```
IF new link state is UP
        THEN CALL Restore Credit Transactions
                remove any previously transmitted message, if any, from TX LIST
                start transmission of next message, if any, from TX LIST
ELSIF new link state  is DOWN
        THEN CALL Save Credit Transactions
ELSIF new link state is NO CHANGE
        THEN remove any previously transmitted message, if any, from TX LIST
                start transmission of next message, if any, from TX LIST
```

```
ENDIF
```

2.5.2 Save Credit Transactions

```
FOR EVERY message on TX LIST
    IF message is a Credit Transaction
        THEN Save message to CREDIT TRANSACTION LIST on  Disk
    ENDIF
    Remove message from TX LIST
END FOR EVERY message on TX LIST LOOP
```

2.5.3 Restore Credit Transactions

```
FOR EVERY message on CREDIT TRANSACTION LIST on Disk
    Add message to head of TX LIST
    Remove message from CREDIT TRANSACTION LIST on Disk
END FOR EVERY message on CREDIT TRANSACTION List on Disk LOOP
```

3.2 Monitor Sensors

```
LOOP FOREVER UNTIL Disabled from Gas Station STD
    READ sensor data
    convert the sensor data to processed sensor data (i.e., a REAL value)
    compute delta from Threshold using processed sensor data and Threshold
    IF previous delta was below Threshold and new delta is above Threshold
        THEN Send Threshold Exceeded Event to Gas Station STD
    ENDIF
END FOREVER LOOP
```

3.3 Sound Alarm

```
Send TURN ON COMMAND to Alarm
```

3.4 Reset Alarm

```
Send TURN OFF COMMAND to Alarm
```

3.5 Send Alarm Message

```
Alarm Message := BUILD(Remote Central Facility Address, Station ID)
CALL Add to Tx List(Alarm Message)
```

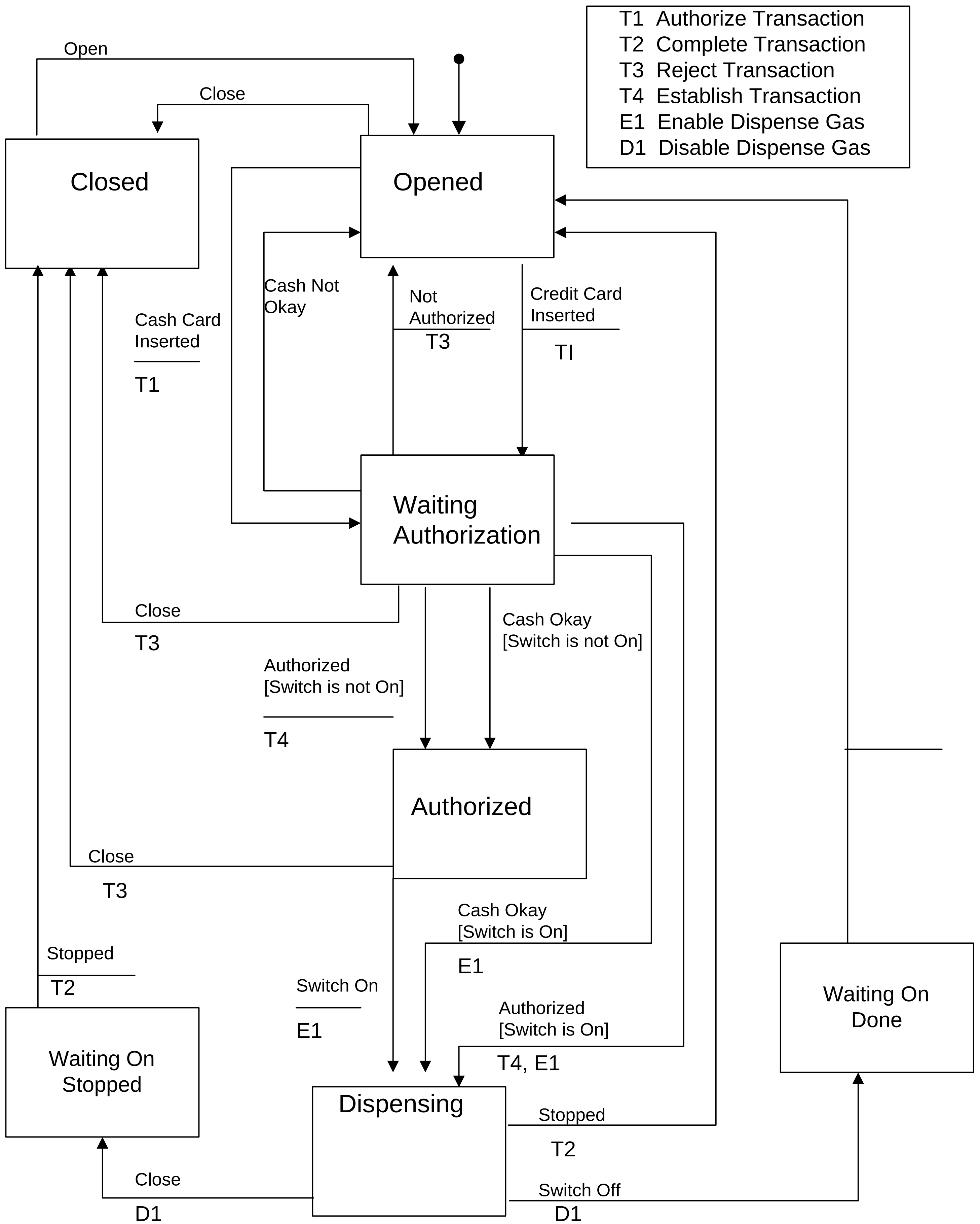
3.6 Send Close

```
LOOP FOR EVERY Pump
    Send Close to Pump
END LOOP
```

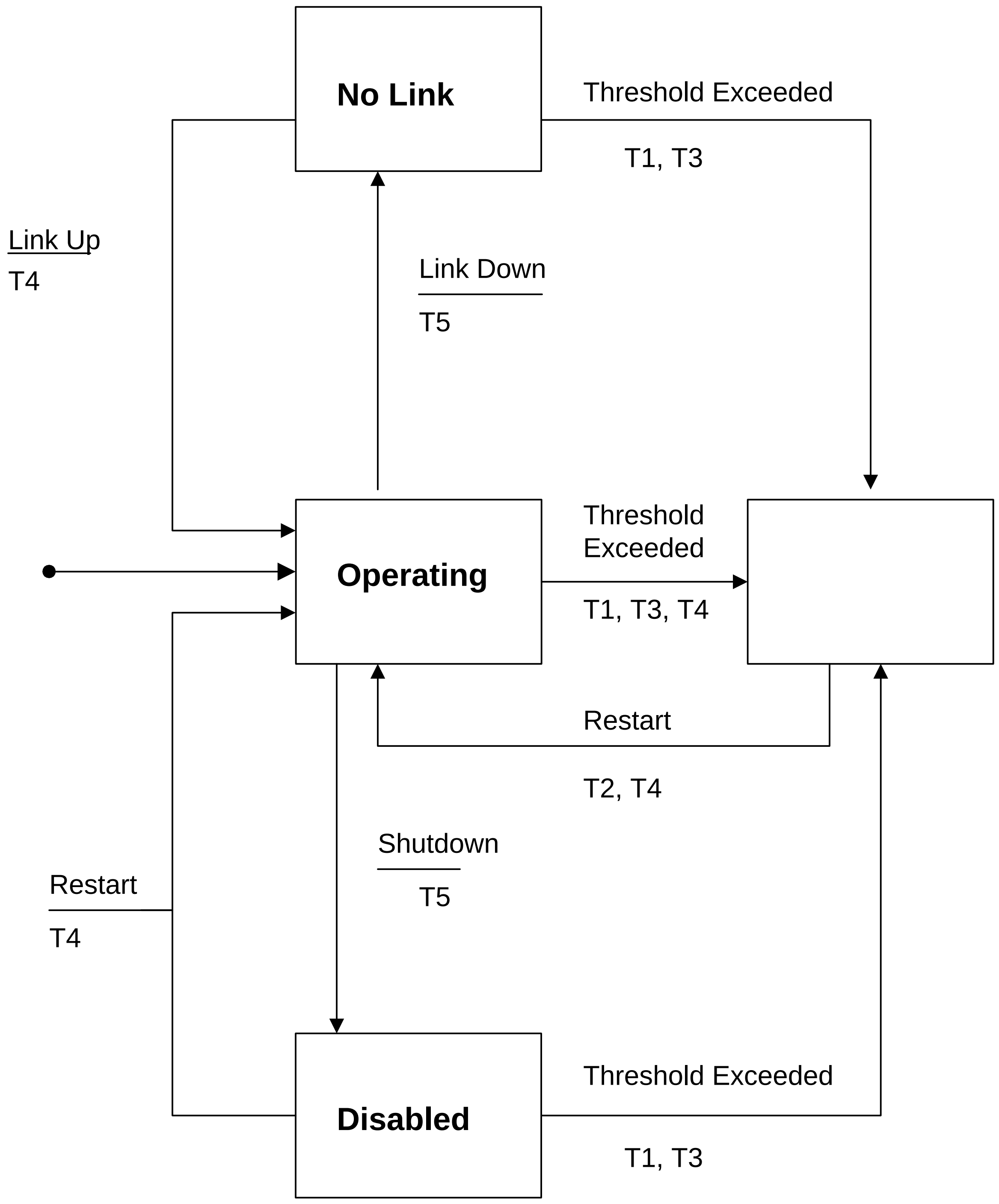
3.7 Send Open

```
LOOP FOR EVERY PUMP
```

```
    Send Open to Pump
END LOOP
```



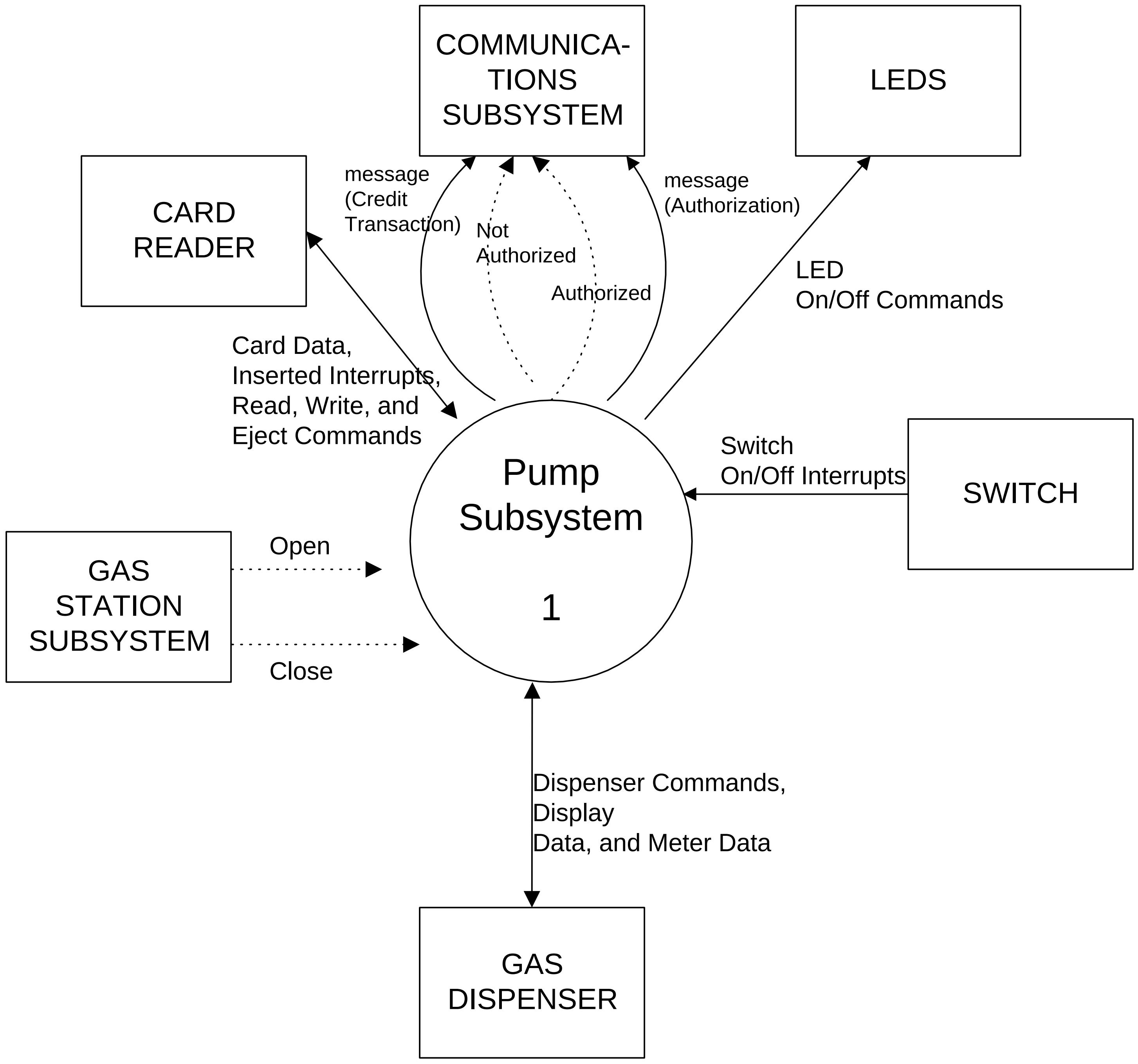
B-19 1.1 Control Pump STATE TRANSITION DIAGRAM



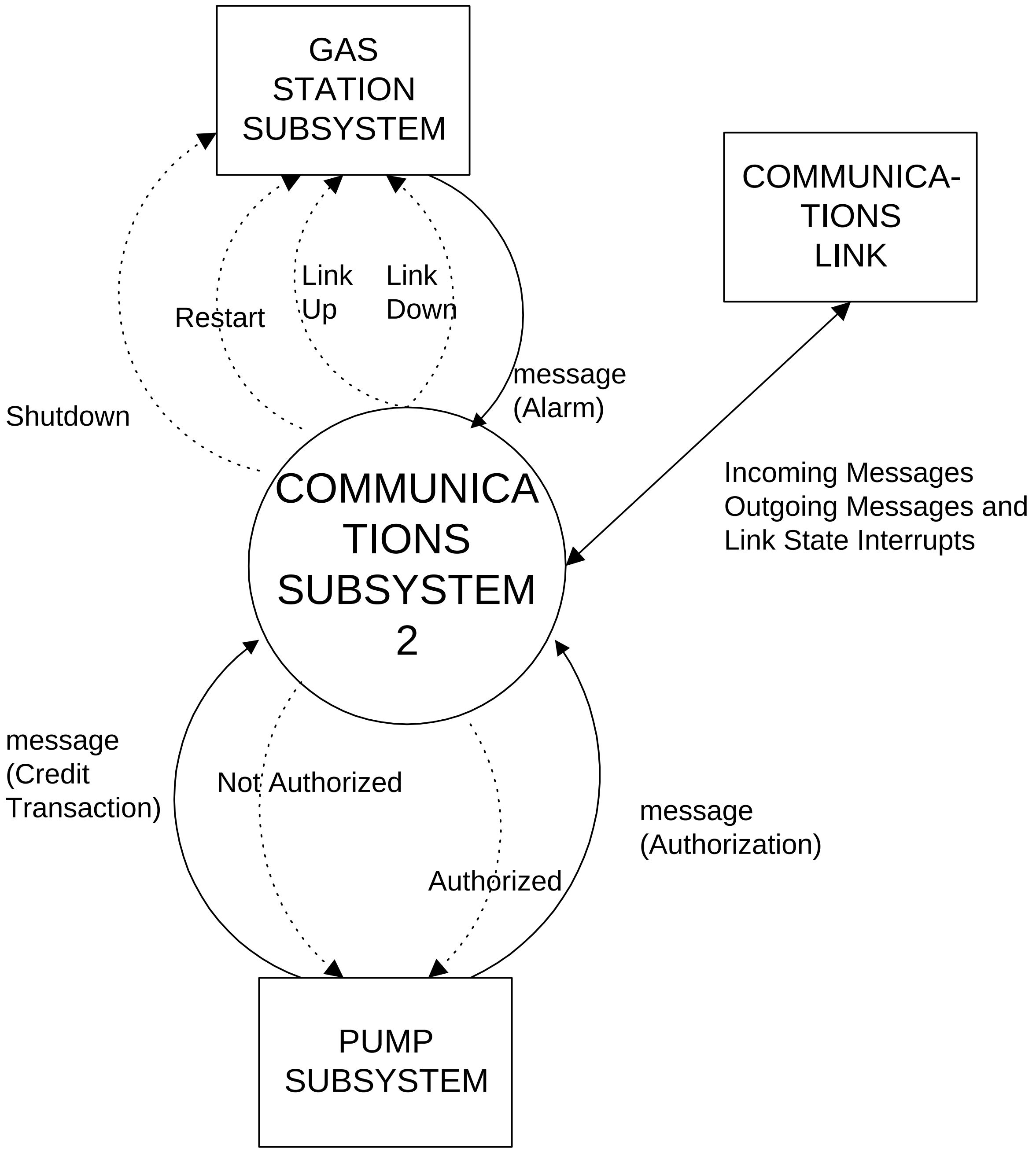
T1 Sound Alarm
T2 Reset Alarm
T3 Send Alarm Message
T4 Send Opens
T5 Send Closes

B-20 3.1 CONTROL GAS STATION STATE TRANSITION DIAGRAM

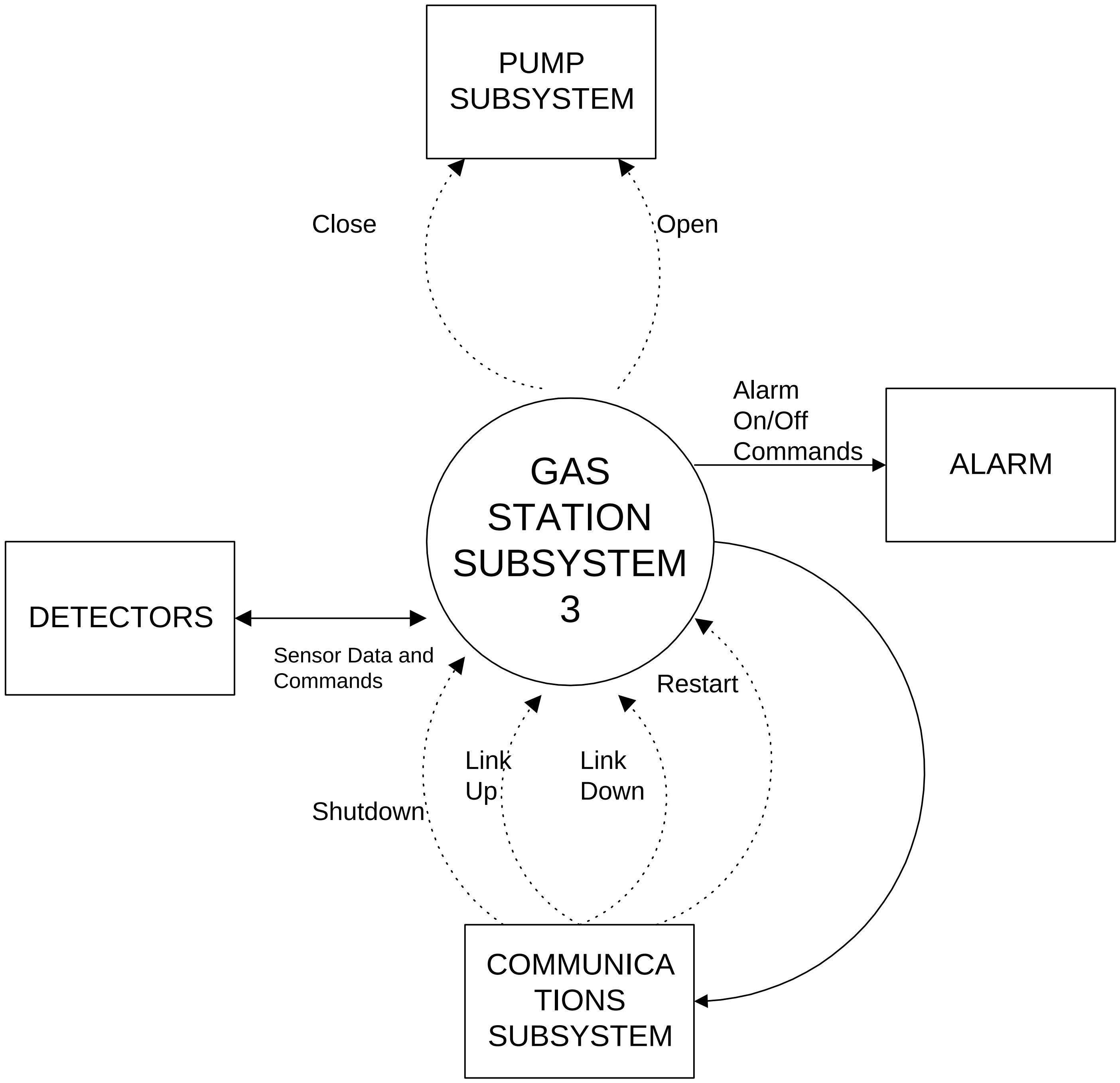
APPENDIX C. AUTOMATED GAS STATION MANAGER COBRA SPECIFICATION



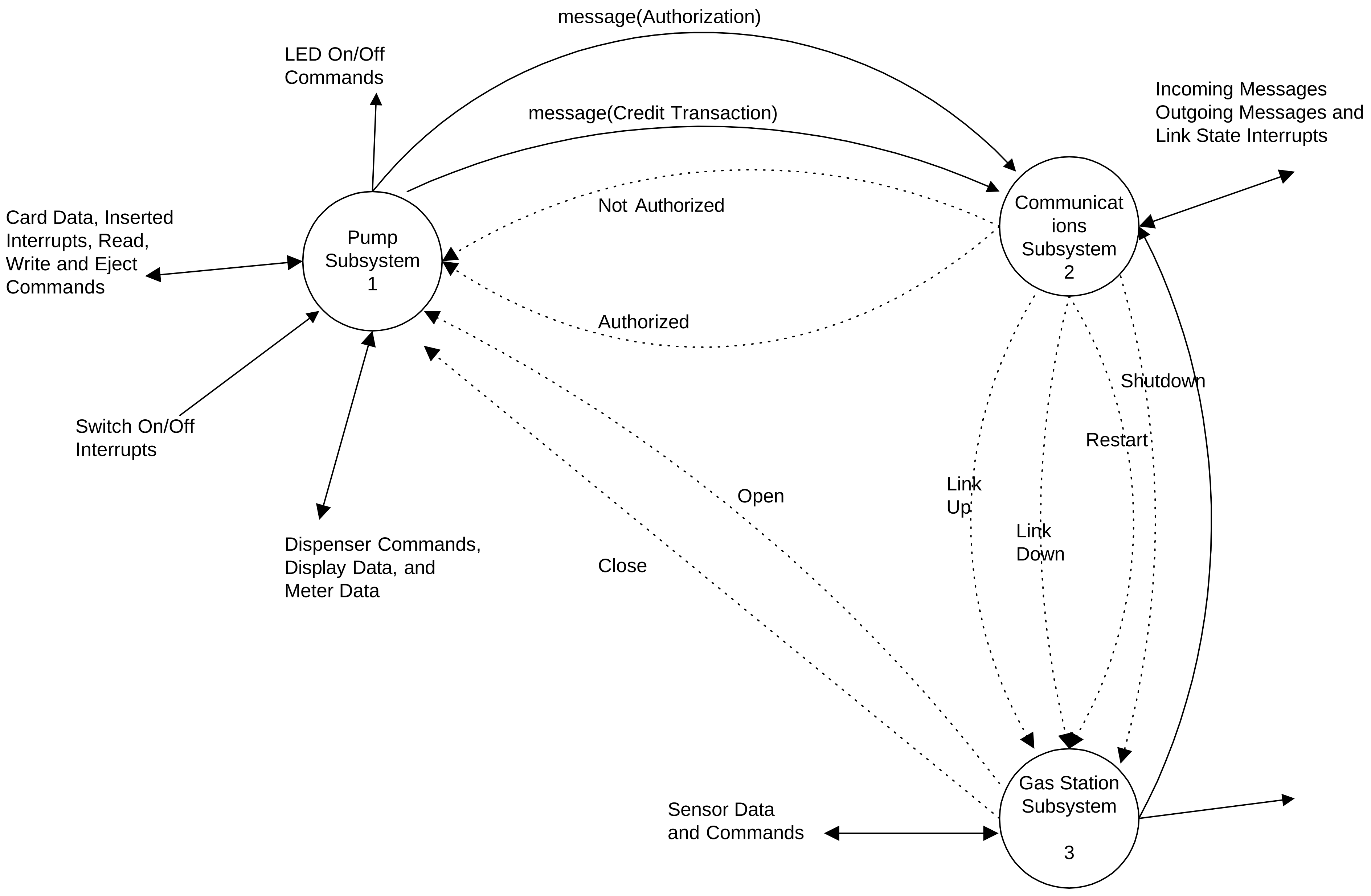
C-4 PUMP SUBSYSTEM CONTEXT DIAGRAM



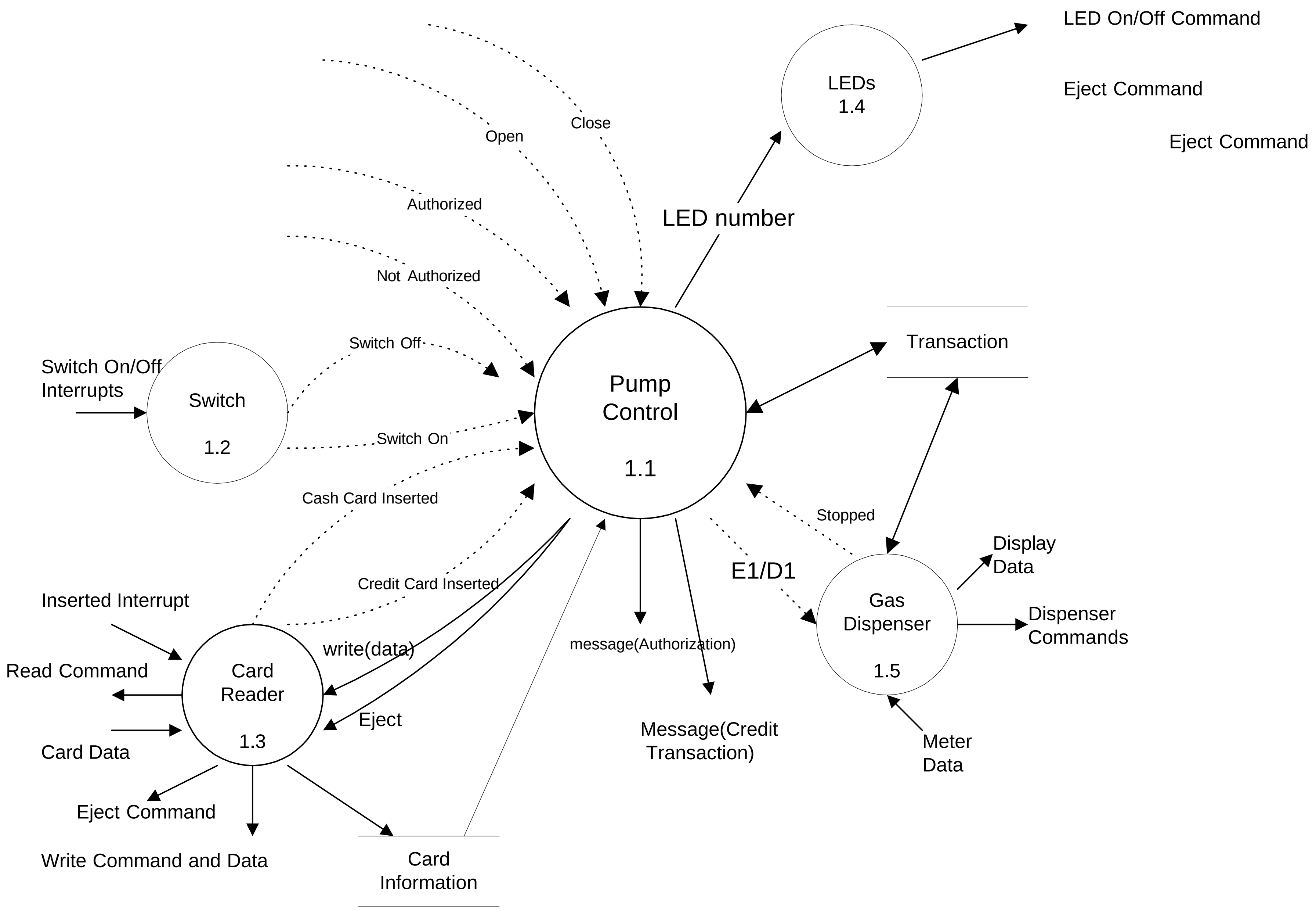
C-5 COMMUNICATIONS SUBSYSTEM CONTEXT DIAGRAM



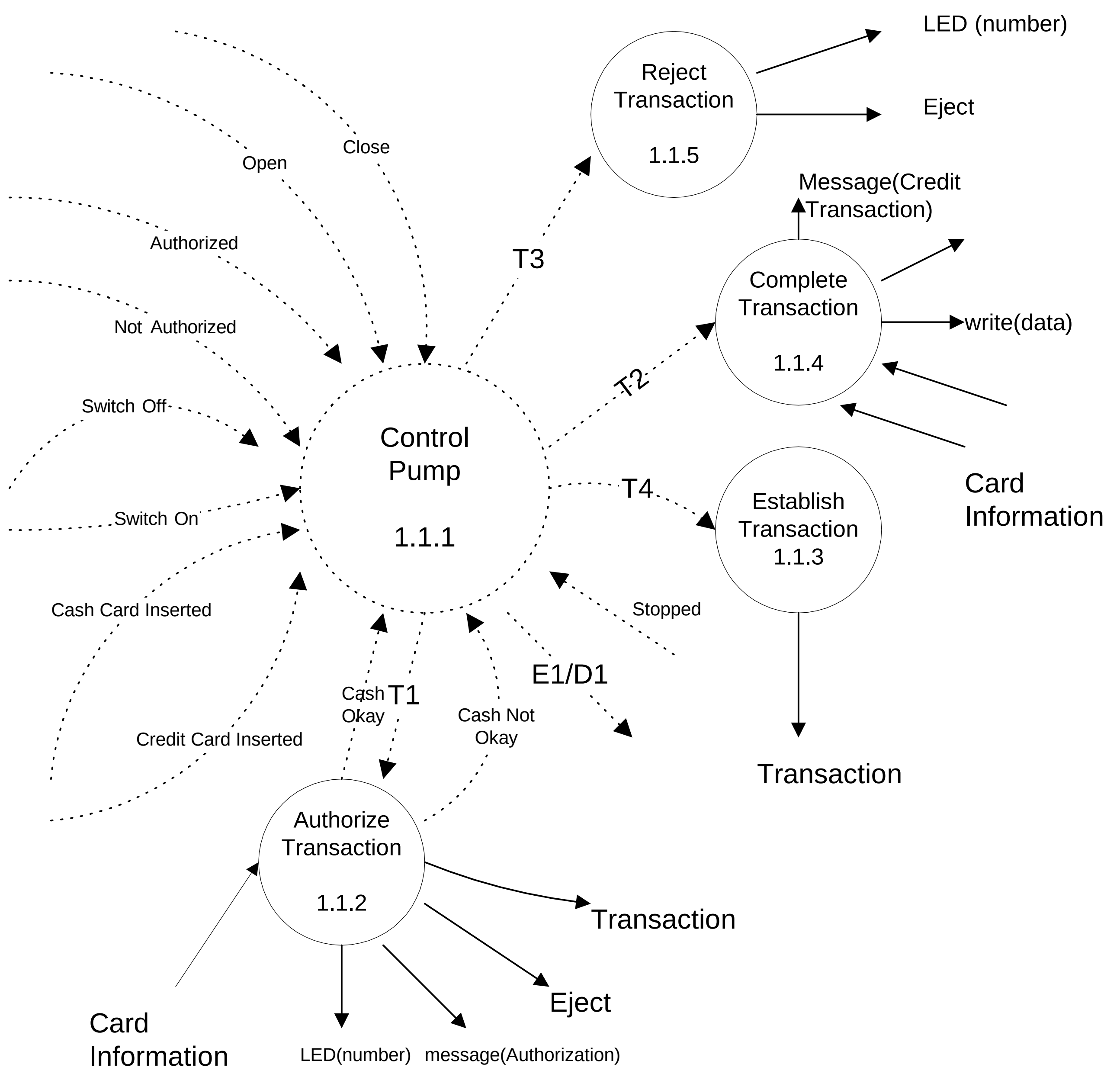
C-6 GAS STATION SUBSYSTEM CONTEXT DIAGRAM


message(Authorization)
LED On/Off Commands
message(Credit Transaction)
Incoming Messages Outgoing Messages and Link State Interrupts
Card Data, Inserted Interrupts, Read, Write and Eject Commands
Pump Subsystem 1
Not Authorized
Communications Subsystem 2
Authorized
Shutdown
Switch On/Off Interrupts
Restart
Link Up
Open
Link Down
Dispenser Commands, Display Data, and Meter Data
Close
Gas Station Subsystem 3
Sensor Data and Commands

C-9 SUBSYSTEM LEVEL D/CFD



C-9 PUMP SUBSYSTEM TOP-LEVEL DATA/CONTROL FLOW DIAGRAM



C-10 1.1 Pump Control DATA/CONTROL FLOW DIAGRAM

PUMP SUBSYSTEM DATA DICTIONARY

Card Data : RECORD

-- Read from an inserted Card

1. identification number : ASCII STRING	-- Determines second field
2. account number : ASCII STRING	-- If Credit Card
3. cash value : ASCII STRING	-- If Cash Card

Card Information: RECORD

--Holds the data read from a Card inserted into the Card Reader

1. identification number: NUMERIC	
2. account number: NUMERIC	-- Valid for Credit Card Only
3. cash value: INTEGER	-- Valid for Cash Card Only

Card Reader Commands

0 is Eject	
1 is Read	(Input Data is a Card Data Record)
2 is Write	(Output Data is a Card Data Record)

Dispenser Commands

0 is STOP GAS	
1 is START GAS	
2 is READ METER	(Input data is Meter Data)
3 is WRITE DISPLAY	(Output data is Display Data)

Display Data: RECORD

--Contains two values to be displayed

1. display gallons: FIXED POINT (Gallons and Tenths and Hundreths)
2. display cost: FIXED POINT (Dollars and Cents)

LED(number) : INTEGER

0 is Cash Value Used
1 is Cannot Process Card

Message: RECORD

--The Data Structure that holds a message

1. To: ADDRESS (Station or Remote Central Facility)
2. From: ADDRESS (Station or Remote Central Facility)
3. Type: BYTE
   - 2 is Credit Transaction
   - 4 is Authorization request
4. Optional additional parameters determined by Type.
   - For Credit Transaction:
     - Pump ID: INTEGER
     - account number: NUMERIC
     - cost of gas: INTEGER (in cents)
   - For Authorization Request
     - Pump ID: INTEGER
     - account number: NUMERIC

Meter Data: REAL

-- Data read from the Gas Dispenser's meter

Transaction : RECORD

1. STATION ID : INTEGER
2. PUMP ID : INTEGER
3. cost of gas : INTEGER
4. limit : INTEGER

Pump Subsystem Mini-Specifications

1.1.2 Authorize Transaction

```
IF Card Information.identification number is for a Cash Card
    THEN
        IF cash value is not positive
            THEN CALL LED(Cash Value Used)
                 Send Cash Not Okay Event to Pump STD
                 Wait 10 Seconds
                 Call Card Reader.Eject
            ELSE Create Transaction with STATION ID and PUMP ID
                 Set Transaction.limit to cash value
                 Send Cash Okay Event to Pump STD
        ENDIF
    ELSE Authorization Request := BUILD (Remote Central Facility Address, Station ID,
                 Pump ID, Card Information.account number)
        CALL Add to Tx List(Authorization Request)
ENDIF
```

1.1.3 Establish Transaction

```
Create Transaction with STATION ID and PUMP ID
Set Transaction.limit to zero
```

1.1.4 Complete Transaction

```
IF Card Information.identification number is for Credit Card
    THEN Credit Transaction := BUILD(Remote Central Facility Address, Station ID, Pump
                                     ID, account number,
                                     amount dispensed/100*price per gallon)
         CALL Add to TX List (Credit Transaction)
    ELSE new cash value := cash limit - (amount dispensed/100*price per gallon)
         CALL Card Reader.write(Card Information.identification number, new cash
                                     value)
         CALL Card Reader.Eject
ENDIF
```

1.1.5 Reject Transaction

```
CALL LED(Cannot Process Card)
CALL Card Reader.Eject
```

1.2 Switch

```
LOOP FOREVER
WAIT for ACTIVATE INTERRUPT OR DEACTIVATE INTERRUPT
```

```
IF ACTIVATE INTERRUPT
    THEN Send Switch ON to Pump STD
    ELSE  Send Switch OFF to Pump STD
ENDIF
END  FOREVER LOOP
```

1.3 Card Reader

```
LOOP FOREVER A
    LOOP FOREVER B
        WAIT for CARD INSERTED INTERRUPT
        READ identification number and store in Card Information
        IF identification number is Cash Card
            THEN READ cash value,
                store cash value in Card Information,
                Send Cash Card Inserted Event to Pump STD
                BREAK LOOP B
        ELSEIF identification number is Corporate Credit Card
            THEN READ account number,
                store account number in Card Information,
                Send Credit Card Inserted Event to Pump STD
                BREAK LOOP B
        ELSE  OUTPUT EJECT COMMAND to Card Reader
        ENDIF
    END LOOP B
    LOOP FOREVER C
        WAIT for Card Reader Request
        IF Request is Eject THEN OUTPUT EJECT COMMAND to Card Reader
                                 BREAK LOOP C
        IF Request is write with data THEN OUTPUT WRITE COMMAND with
                                           data to Card Reader
    END LOOP C
END LOOP A
```

1.4 LEDs

```
IF LED(number) is Cash Value Used
    THEN OUTPUT LED ON COMMAND to LED #1
         WAIT 10 Seconds
         OUTPUT LED OFF COMMAND to LED #1
ELSIF LED(number) is Could Not Process Card
    THEN OUTPUT LED ON COMMAND to LED #2
         WAIT 10 Seconds
         OUTPUT LED OFF COMMAND to LED #2
ENDIF
```

1.5 Gas Dispenser

```
Send START COMMAND to Gas Dispenser
```

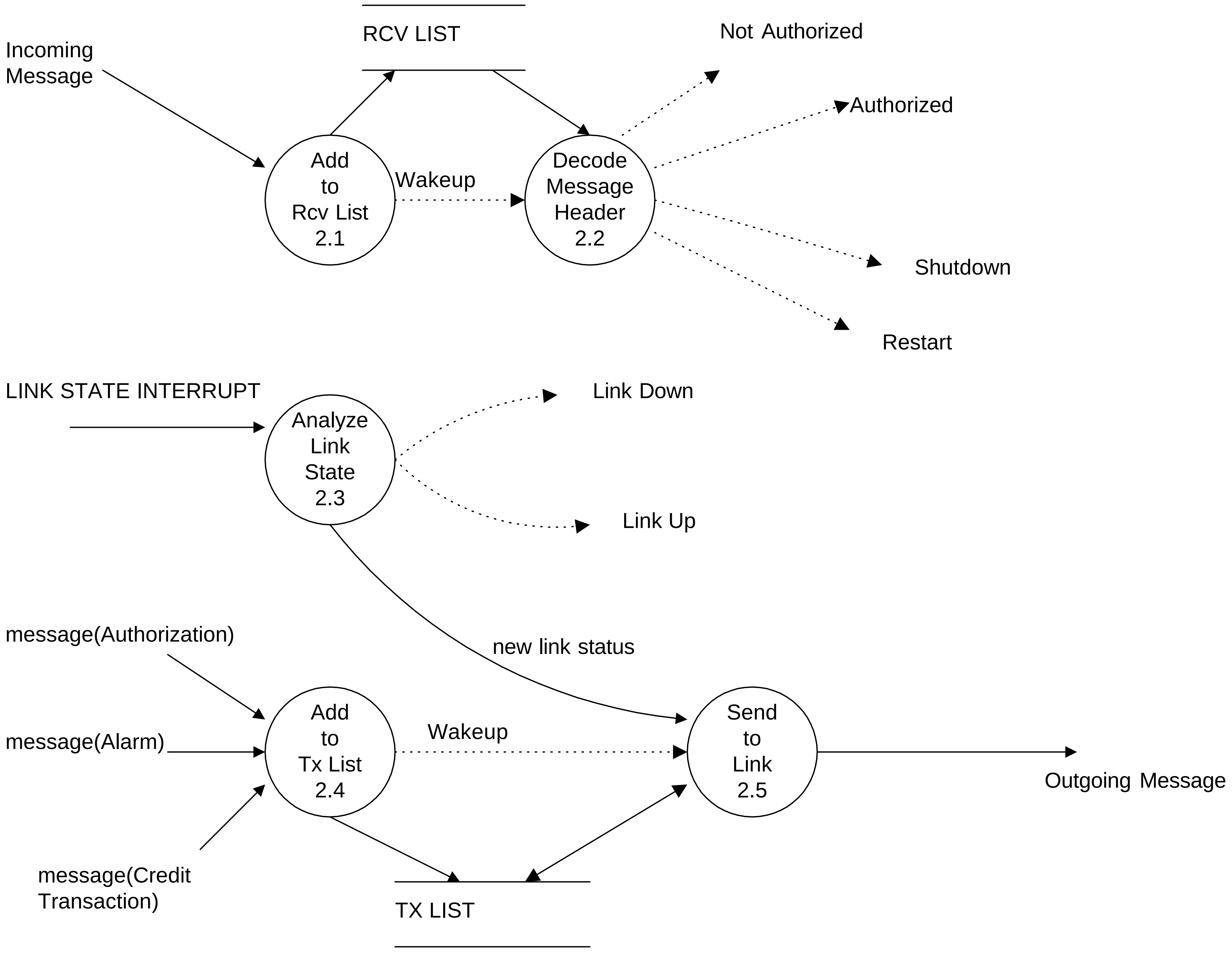
```
LOOP FOREVER UNTIL Disabled from Pump STD
    CALL Monitor Meter
    CALL Update Display
    If amount dispensed/100 * price per gallon
        >= cash limit THEN BREAK LOOP
END FOREVER LOOP
Send STOP COMMAND to Gas Dispenser
CALL Monitor Meter
CALL Update Display
Set Transaction.cost of gas to amount dispensed/100 * price per gallon
Send Stopped to Pump STD
RETURN
```

Monitor Meter

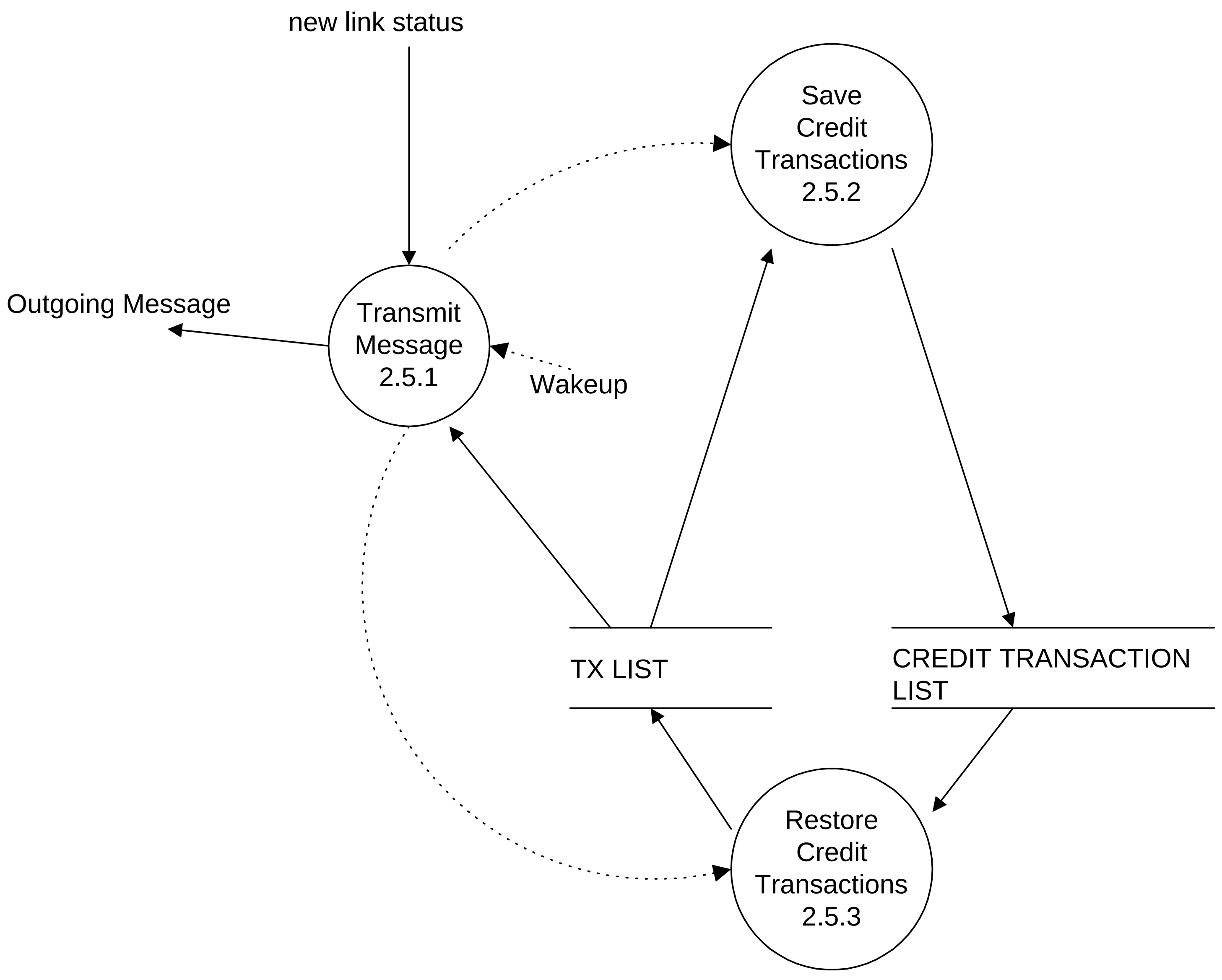
```
READ meter value
Convert to hundreths of gallons
Set amount dispensed to converted value
```

Update Display

```
display gallons := amount dispensed/100
display cost := display gallons * price per gallon
Output display gallons to GALLONS DISPLAY
Output display cost to COST DISPLAY
```



B-5 2 MANAGE COMMUNICATIONS LINK DATA FLOW DIAGRAM



B-6 2.5 Send to Link DATA FLOW DIAGRAM

Communications Subsystem Data Dictionary

CREDIT TRANSACTION LIST: FILE

--A File containing saved Credit Transaction Messages

Message: RECORD

--The Data Structure that holds a message

1. To: ADDRESS (Station or Remote Central Facility)
2. From: ADDRESS (Station or Remote Central Facility)
3. Type: BYTE
    - 0 is SHUTDOWN COMMAND
    - 1 is RESTART COMMAND
    - 2 is Credit Transaction
    - 3 is Alarm
    - 4 is Authorization request
    - 5 is Authorization reply
4. Optional additional parameters determined by Type.
    - For Credit Transaction:
        - Pump ID: INTEGER
        - account number: NUMERIC
        - cost of gas: INTEGER (in cents)
    - For Authorization Request
        - Pump ID: INTEGER
        - account number: NUMERIC
    - For Authorization Reply
        - Pump ID: INTEGER
        - response: BOOLEAN
            - 0 is Not Authorized
            - 1 is Authorized

new link status: INTEGER

-- The new status of the communications link. Values can be 0, no change, 1, link up, or
-- -1, link down

RCV LIST: LINKED_LIST

-- This is the queue of messages received on the Communications Link, but not yet
-- processed.

TX LIST: LINKED_LIST

-- The queue of messages that are to be transmitted, but that have not yet been transmitted

Communications Subsystem Mini-Specifications

2.1 Add to Rcv List

```
Accept message from Link
Add message to RCV LIST
CALL Decode Message Header
```

2.2 Decode Message Header

```
IF message is a RESTART COMMAND
        THEN generate a Restart event for the Gas Station STD
ELSEIF message is a SHUTDOWN COMMAND
        THEN generate a Shutdown event for the Gas Station STD
ELSEIF message is an authorization reply
        THEN IF message contains authorization
                        THEN generate an Authorized event for the appropriate
                                        Pump STD
                        ELSE generate an Unauthorized event for the appropriate
                                        Pump STD
                ENDIF
ENDIF
Remove message from the RCV LIST
```

2.3 Analyze Link State

```
IF LINK STATE is UP and was DOWN
   THEN CALL Transmit Message with new link state set to UP
ELSIF LINK STATE is DOWN and was UP
    THEN CALL Transmit Message with new link state set to DOWN
ELSE CALL Transmit Message with new link state set to NO CHANGE
```

2.4 Add to Tx List(message)

```
Add message to TX LIST
CALL Transmit Message(NO CHANGE)
```

2.5.1 Transmit Message(new link state)

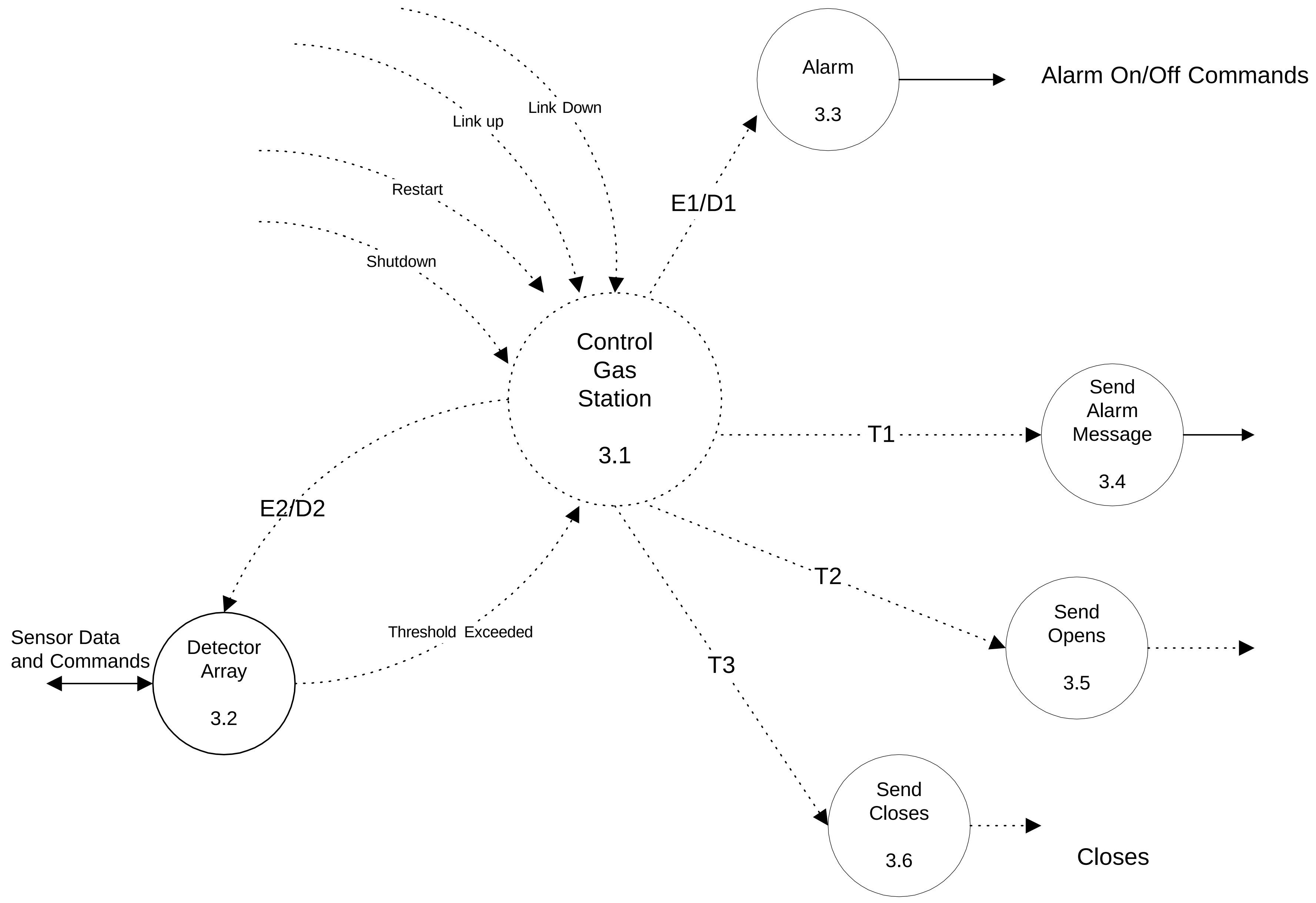
```
IF new link state is UP
        THEN CALL Restore Credit Transactions
                remove any previously transmitted message, if any, from TX LIST
                start transmission of next message, if any, from TX LIST
ELSIF new link state  is DOWN
        THEN CALL Save Credit Transactions
ELSIF new link state is NO CHANGE
        THEN remove any previously transmitted message, if any, from TX LIST
                start transmission of next message, if any, from TX LIST
ENDIF
```

2.5.2 Save Credit Transactions

```
FOR EVERY message on TX LIST
    IF message is a Credit Transaction
        THEN Save message to CREDIT TRANSACTION LIST on  Disk
    ENDIF
    Remove message from TX LIST
END FOR EVERY message on TX LIST LOOP
```

2.5.3 Restore Credit Transactions

```
FOR EVERY message on CREDIT TRANSACTION LIST on Disk
    Add message to head of TX LIST
    Remove message from CREDIT TRANSACTION LIST on Disk
END FOR EVERY message on CREDIT TRANSACTION List on Disk LOOP
```



C-27 3 GAS STATION SUBSYSTEM TOP-LEVEL CONTROL/DATA FLOW DIAGRAM

Gas Station Subsystem Data Dictionary

Alarm Commands

0 is TURN OFF
1 is TURN ON

Message: RECORD

--The Data Structure that holds a message

1. To: ADDRESS (Station or Remote Central Facility)
2. From: ADDRESS (Station or Remote Central Facility)
3. Type: BYTE
   3 is Alarm
4. Optional additional parameters determined by Type.

Sensor Commands

0 is Enable
1 is Disable
2 is Read (Input Data is Sensor Data)

Sensor Data: DEVICE DEPENDENT

-- The raw input from a sensor device. The form is dependent on the specific sensor.

Gas Station Subsystem Mini-Specifications

3.2 Detector Array

```
WHEN Enabled
    LOOP UNTIL Disabled from Gas Station STD
        READ sensor data
        convert the sensor data to processed sensor data (i.e., a REAL value)
        compute delta from Threshold using processed sensor data and Threshold
        IF previous delta was below Threshold and new delta is above Threshold
            THEN Send Threshold Exceeded Event to Gas Station STD
        ENDIF
    END  LOOP
```

3.3 Alarm

```
WHEN Enabled
    LOOP UNTIL Disabled from Gas Station STD
        OUTPUT ON COMMAND to Alarm
    END LOOP
OUTPUT OFF COMMAND to Alarm
```

3.4 Send Alarm Message

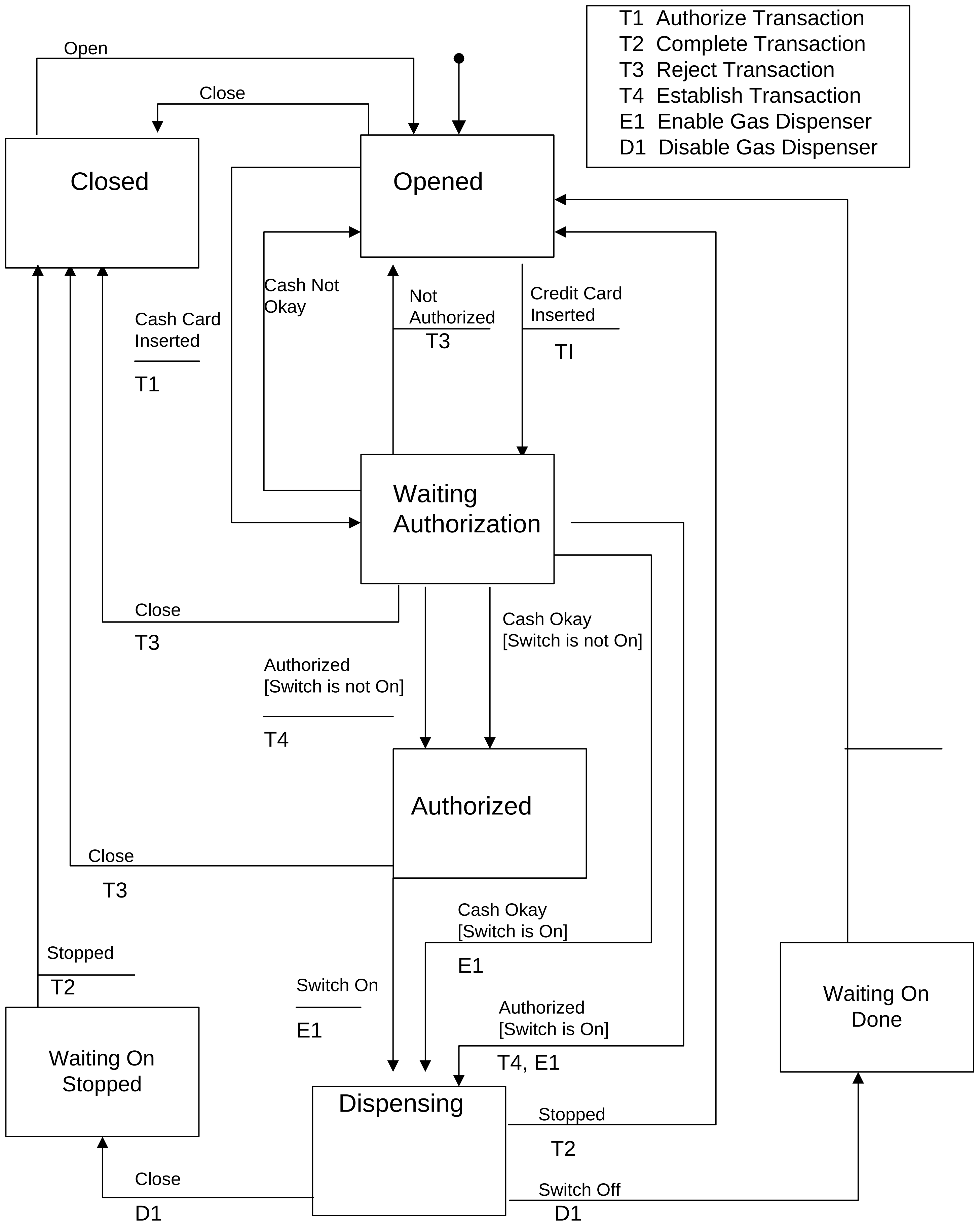
```
Alarm Message := BUILD(Remote Central Facility Address, Station ID)
CALL Add to Tx List(Alarm Message)
```

3.5 Send Close

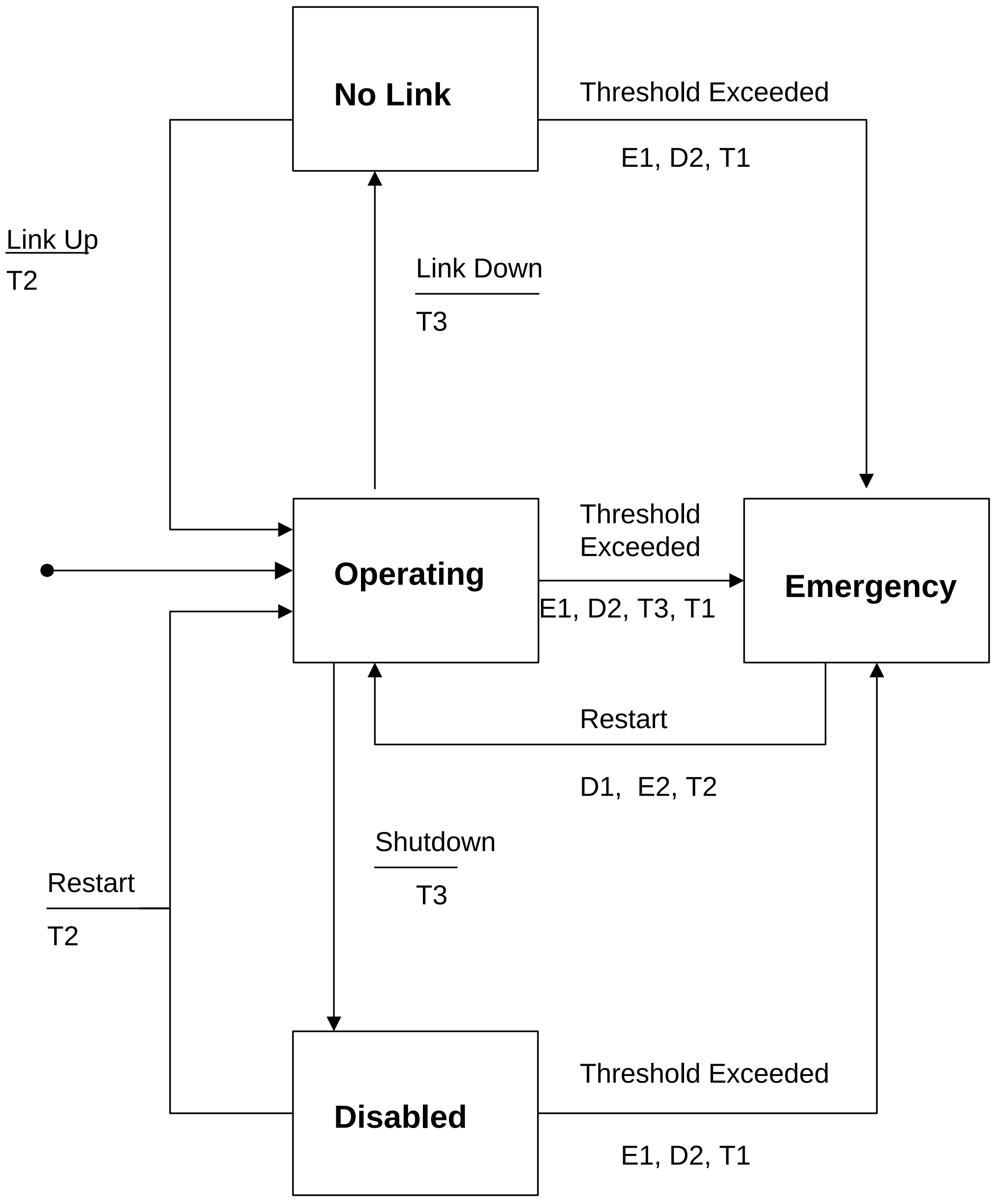
```
LOOP FOR EVERY Pump
    Send Close to Pump
END LOOP
```

3.6 Send Open

```
LOOP FOR EVERY PUMP
    Send Open to Pump
END LOOP
```



C-16 1.1 CONTROL PUMP STATE TRANSITION DIAGRAM



T1 Send Alarm Message
T2 Send Opens
T3 Send Closes
E1 Enable Alarm
D1 Disable Alarm

C-30 3.1 Control Gas Station STATE TRANSITION DIAGRAM

AGMS Scenario Event Descriptions

What follows are narrative descriptions of each of the twenty-one scenarios depicted in the following figures on pages C-32 through C-52. These scenarios were used to develop the state transition diagrams for the Pump (C-53) and the Gas Station (C-54).

Scenario #1 - Exhausted Cash Card Inserted By The Customer (C-32)

The customer inserts a cash card into the card reader causing an Inserted Interrupt (S1-1). The card reader reads the data from the cash card, stores that information (S1-2) and then generates a Cash Card Inserted (S1-3) event for the Control Pump STD. Control Pump triggers (S1-4) a function to authorize the cash transaction. Authorize Transaction examines the Card Information (S1-5), sees that the cash card is exhausted, lights the Cash Value Used LED (S1-6), ejects the cash card (S1-7), and generates a Cash Not Okay (S1-8) event for the Control Pump STD.

Scenario #2 - Valid Cash Card Inserted By The Customer (C-33)

The customer inserts a cash card into the card reader causing an Inserted Interrupt (S2-1). The card reader reads the data from the cash card, stores that information (S2-2) and then generates a Cash Card Inserted (S2-3) event for the Control Pump STD. Control Pump triggers (S2-4) a function to authorize the cash transaction. Authorize Transaction examines the Card Information (S2-5), sees that the cash card is valid, creates a cash transaction (S2-6), generates a Cash Okay (S2-7) event for the Control Pump STD. Then, if the pump switch is ON, the Control Pump STD enables the Gas Dispenser (S2-8).

Scenario #3 - Corporate Credit Card Inserted By The Customer (C-34)

The customer inserts a corporate credit card into the card reader causing an Inserted Interrupt (S3-1). The card reader reads the data from the credit card, stores that information (S3-2) and then generates a Credit Card Inserted (S3-3) event for the Control Pump STD. Control Pump triggers (S3-4) a function to authorize the credit transaction. Authorize Transaction extracts the account number from the Card Information (S3-5) and then sends an Authorization Request message to the Remote Central Facility (S3-6).

Scenario #4 - Gas Dispenser Switch Set To On By Customer After The Transaction Has Been Authorized (C-35)

The customer moves the pump switch to the ON position, generating a Switch On Interrupt (S4-1) at the Switch object. The Switch object then generates a Switch On event (S4-2) for the Control Pump STD. If the customer’s transaction has already been authorized, then the Control Pump STD enables the Gas Dispenser (S4-3). The Gas Dispenser turns on the dispenser mechanism (S4-4) and then cycles through its dispensing operations, reading the Meter Data (S4-5) and writing the Display Data (S4-6).

Scenario #5 - Gas Dispenser Switch Set To On By Customer Before The Transaction Has Been Authorized (C-36)

The customer moves the pump switch to the ON position, generating a Switch On Interrupt (S5-1) at the Switch object. The Switch object then generates a Switch On event (S5-2) for the Control Pump STD. Because the transaction has not yet been authorized, the Control Pump STD simply remembers that the switch is ON.

Scenario #6 - Gas Dispenser Switch Set To Off By Customer (C-37)

The customer moves the pump switch to the OFF position, generating a Switch Off Interrupt (S6-1) at the Switch object. The Switch object then generates a Switch Off event (S6-2) for the Control Pump STD. If gas was not being dispensed, the Control Pump STD simply remembers that the switch is OFF. If gas was being dispensed, the Control Pump STD disables the Gas Dispenser (S6-3). The Gas Dispenser then turns off the dispenser mechanism (S6-4), reads the final value of the Meter Data (S6-5) and displays the final Display Data (S6-6).

Scenario #7 - Gas Dispenser Stops For Any Reason When A Cash Transaction Was Being Processed (C-38)

The Gas Dispenser generates a Stopped event (S7-1) for the Control Pump STD. The Control Pump STD then triggers a Complete Transaction operation (S7-2). The Complete Transaction operation extracts the cash value from the Card Information (S7-3) and the cost of gas from the Transaction (S7-4), determines the new cash value and writes it to the cash card (S7-5), and then ejects the cash card (S7-6).

Scenario #8 - Gas Dispenser Stops For Any Reason When A Credit Transaction Was Being Processed (C-39)

The Gas Dispenser generates a Stopped event (S8-1) for the Control Pump STD. The Control Pump STD then triggers a Complete Transaction operation (S8-2). The Complete Transaction operation extracts the account number from the Card Information (S8-3) and the cost of gas from the Transaction (S8-4), sends a Credit Transaction message to the Remote Central Facility (S8-5), and then ejects the credit card (S8-6).

Scenario #9 - Incoming Authorization Reply With Authorized, Gas Dispenser Switch Is Not On (C-40)

An incoming message arrives on the communications link (S9-1) and is added to the receive list (S9-2). The decode message function is invoked (S9-3) to extract the message from the receive list (S9-4) and to decode the message and generate an Authorized event (S9-5) for the appropriate Control Pump STD. The Control Pump STD triggers (S9-6) a function to establish a credit transaction (S9-7).

Scenario #10 - Incoming Authorization Reply With Authorized, Gas Dispenser Switch Is On (C-41)

An incoming message arrives on the communications link (S10-1) and is added to the receive list (S10-2). The decode message function is invoked (S10-3) to extract the message from the receive list (S10-4) and to decode the message and generate an Authorized event (S10-5) for the appropriate Control Pump STD. The Control Pump STD triggers (S10-6) a function to establish a credit transaction (S10-7). The Control Pump STD then enables (S10-8) the Gas Dispenser which subsequently turns on the dispenser mechanism (S10-9) and cycles through the dispensing operation, reading the Meter Data (S10-10) and writing the Display Data (S10-11).

Scenario #11 - Incoming Authorization Reply With Not Authorized (C-42)

An incoming message arrives on the communications link (S11-1) and is added to the receive list (S11-2). The decode message function is invoked (S11-3) to extract the message from the receive list (S11-4) and to decode the message and generate a Not Authorized event (S11-5) for the appropriate Control Pump STD. The Control Pump STD triggers (S11-6) a function to reject the transaction. The Reject Transaction function lights the Cannot Process Card LED (S11-7) and ejects the card (S11-8).

Scenario #12 - Incoming Restart When Station Was Previously Shutdown By Remote Central Facility (C-43)

An incoming message arrives on the communications link (S12-1) and is added to the receive list (S12-2). The decode message function is invoked (S12-3) to extract the message from the receive list (S12-4) and to decode the message and generate a Restart event (S12-5) for the Gas Station STD. The Gas Station STD triggers (S12-6) a function to send Open events (S12-7) to each Pump in the station.

Scenario #13 - Incoming Restart When Station Was Previously Shutdown By Emergency (C-44)

An incoming message arrives on the communications link (S13-1) and is added to the receive list (S13-2). The decode message function is invoked (S13-3) to extract the message from the receive list (S13-4) and to decode the message and generate a Restart event (S13-5) for the Gas Station STD. The Gas Station STD disables (S13-7) the Alarm, enables (S13-9) the Detector Array, and triggers (S13-10) a function to send Open events (S13-11) to each Pump in the station.

Scenario #14 - Incoming Shutdown From Remote Central Facility (C-45)

An incoming message arrives on the communications link (S14-1) and is added to the receive list (S14-2). The decode message function is invoked (S14-3) to extract the message from the receive list (S14-4) and to decode the message and generate a Shutdown event (S14-5) for the Gas Station STD. The Gas Station STD triggers (S14-6) a function to send Close events (S14-7) to each Pump in the station.

Scenario #15 - Pump Receives A Close When The Pump Is Idle (C-46)

A Close event (S15-1) arrives at the Control Pump STD. Since the pump is idle, the Control Pump STD simply moves into the closed state.

Scenario #16 - Pump Receives Close While Permission To Dispense Gas Is Pending (C-47)

A Close event (S16-1) arrives at the Control Pump STD. The Control Pump STD triggers (S16-2) a reject transaction operation which lights (S16-3) the Cannot Process Card LED and ejects the card (S16-4).

Scenario #17 - Pump Receives Close While Dispensing Gas (C-48)

A Close event (S17-1) arrives at the Control Pump STD. Since gas was being dispensed, the Control Pump STD disables the Gas Dispenser (S17-2). The Gas Dispenser then turns off the dispenser mechanism (S17-3), reads the final value of the Meter Data (S17-4) and displays the final Display Data (S17-5).

Scenario #18 - Fire Detected When The Gas Station Is Not Operating (C-49)

The Detector Array generates a Threshold Exceeded event (S18-1) for the Gas Station STD. The Gas Station enables (S18-2) the Alarm (which sends an ALARM ON COMMAND to the Alarm (S18-3)), disables (S18-4) the Detector Array, and triggers (S18-5) an operation which sends an Alarm message (S18-6) to the Remote Central Facility.

Scenario #19 - Fire Detected When The Gas Station Is Operating (C-50)

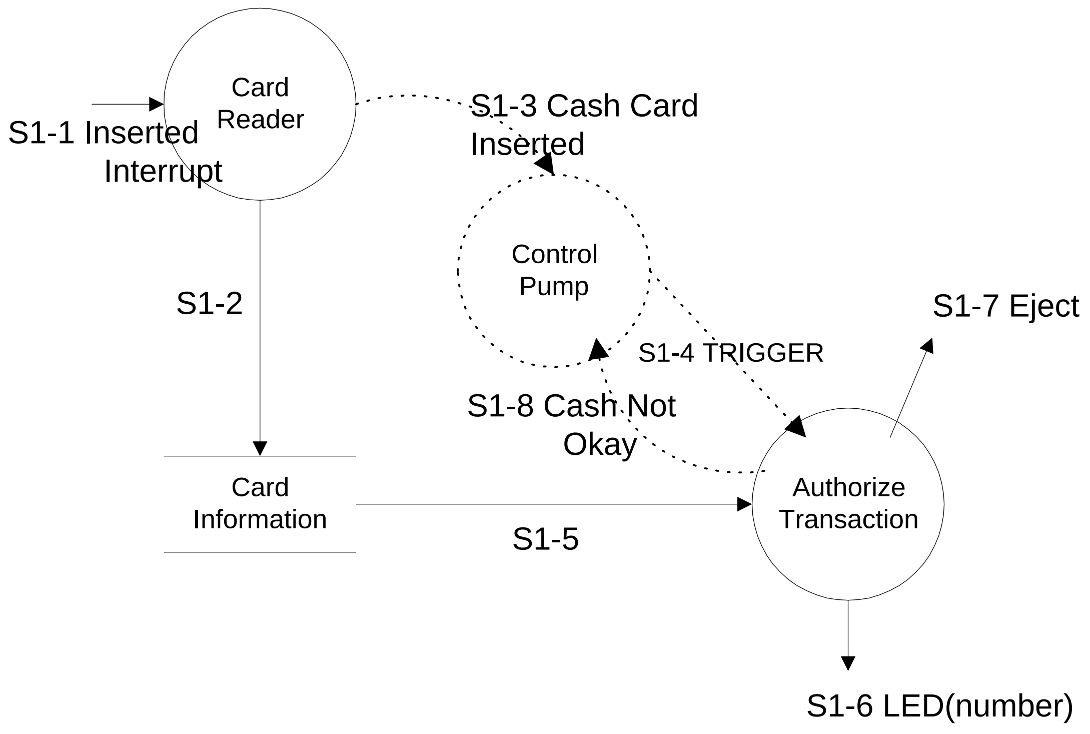
The Detector Array generates a Threshold Exceeded event (S19-1) for the Gas Station STD. The Gas Station enables (S19-2) the Alarm (which sends an ALARM ON COMMAND to the Alarm (S19-3)), disables (S19-4) the Detector Array, triggers (S19-5) an operation which sends Close events (S19-6) to each of the pumps in the gas station, and triggers (S19-7) an operation which sends an Alarm message (S19-8) to the Remote Central Facility.

Scenario #20 - Link State Interrupt Brings The Communications Link Up (C-51)

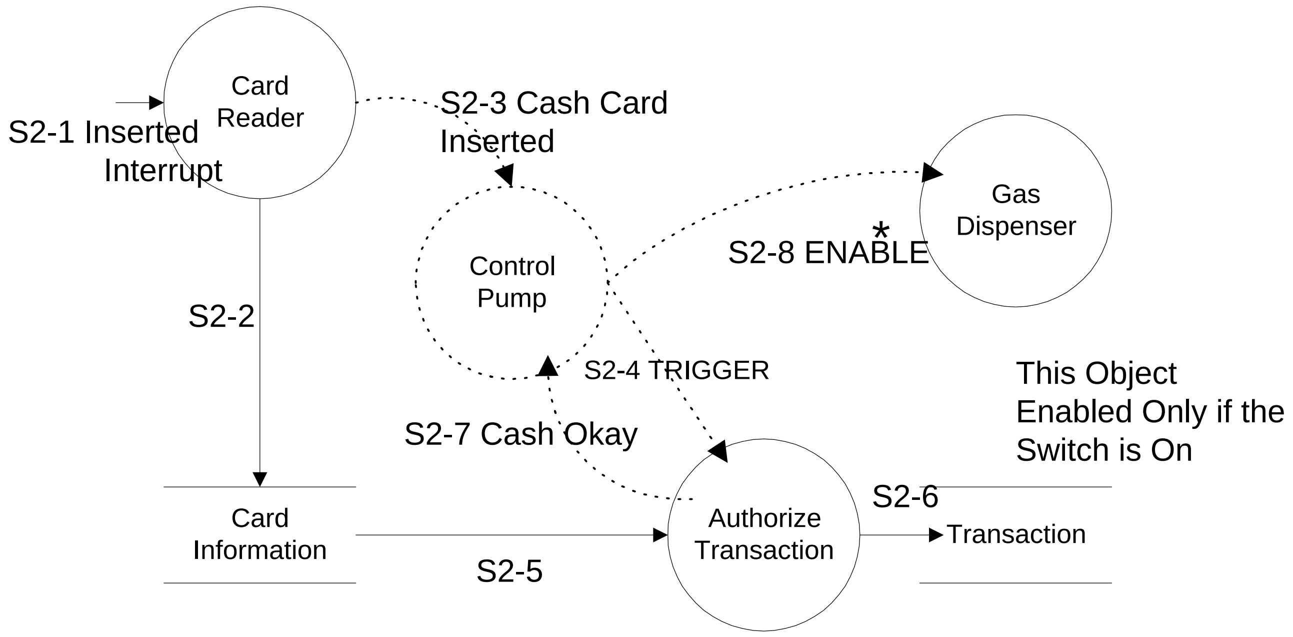
A Link State Interrupt (S20-1) arrives at the Analyze Link State function which, under the proper conditions, generates a Link Up event (S20-2) for the Gas Station STD. The Gas Station STD triggers (S20-3) an operation which sends an Open event (S20-4) to each pump in the gas station.

Scenario #21 - Link State Interrupt Brings The Communications Link Down (C-52)

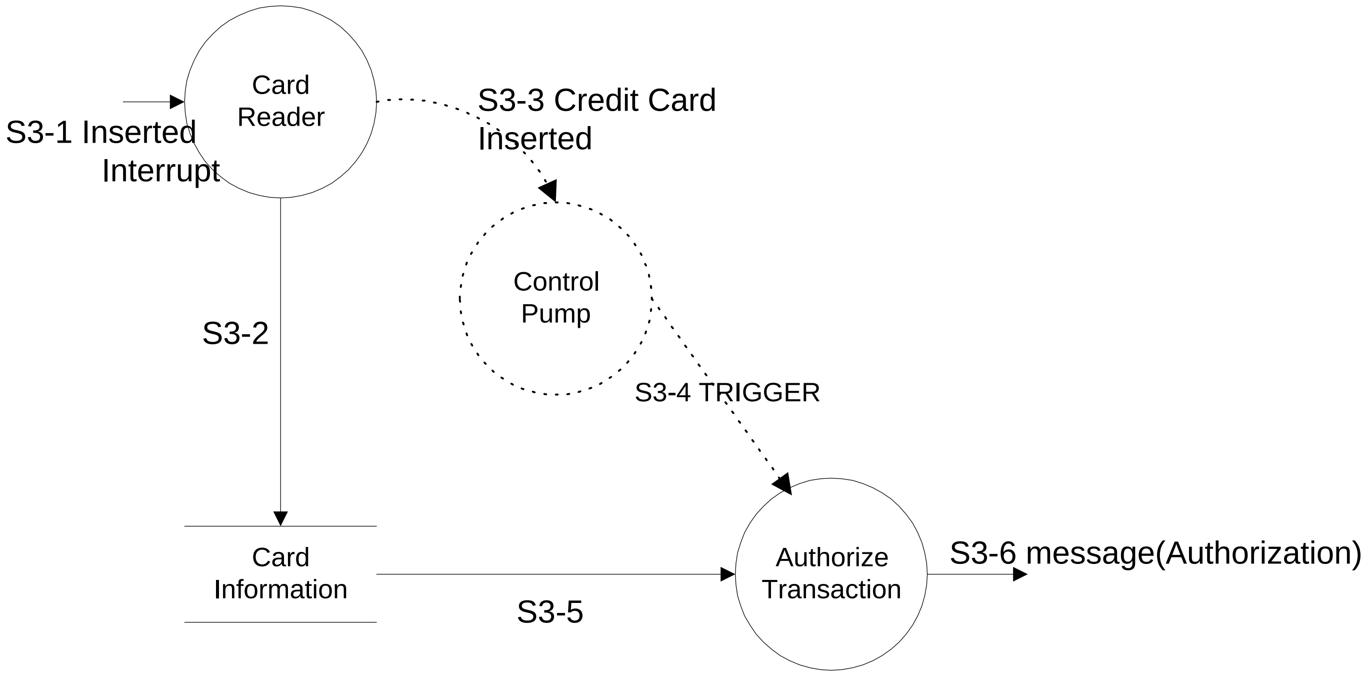
A Link State Interrupt (S21-1) arrives at the Analyze Link State function which, under the proper conditions, generates a Link Down event (S21-2) for the Gas Station STD. The Gas Station STD triggers (S21-3) an operation which sends a Close event (S21-4) to each pump in the gas station.



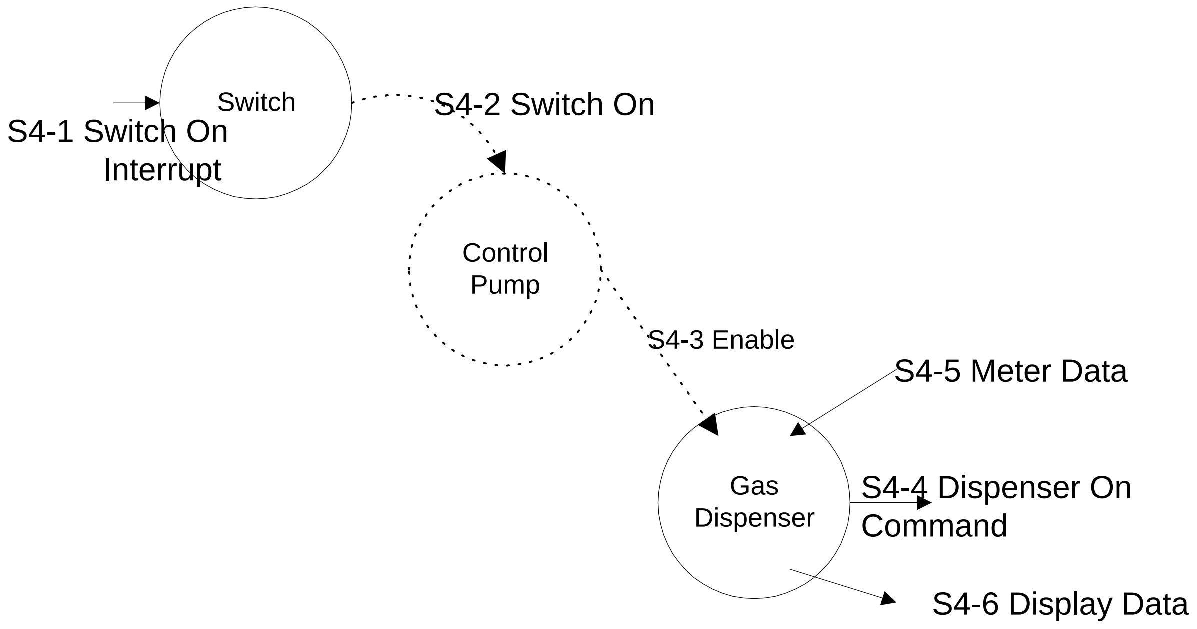
C-32 Scenario #1 - Exhausted Cash Card Inserted By The Customer



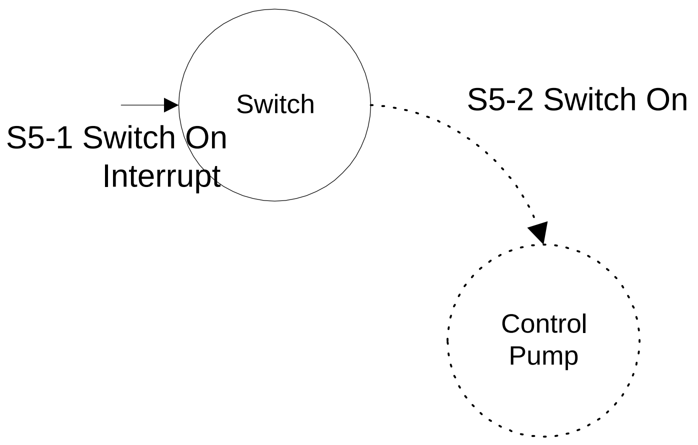
C-33 Scenario #2 - Valid Cash Card Inserted By The Customer



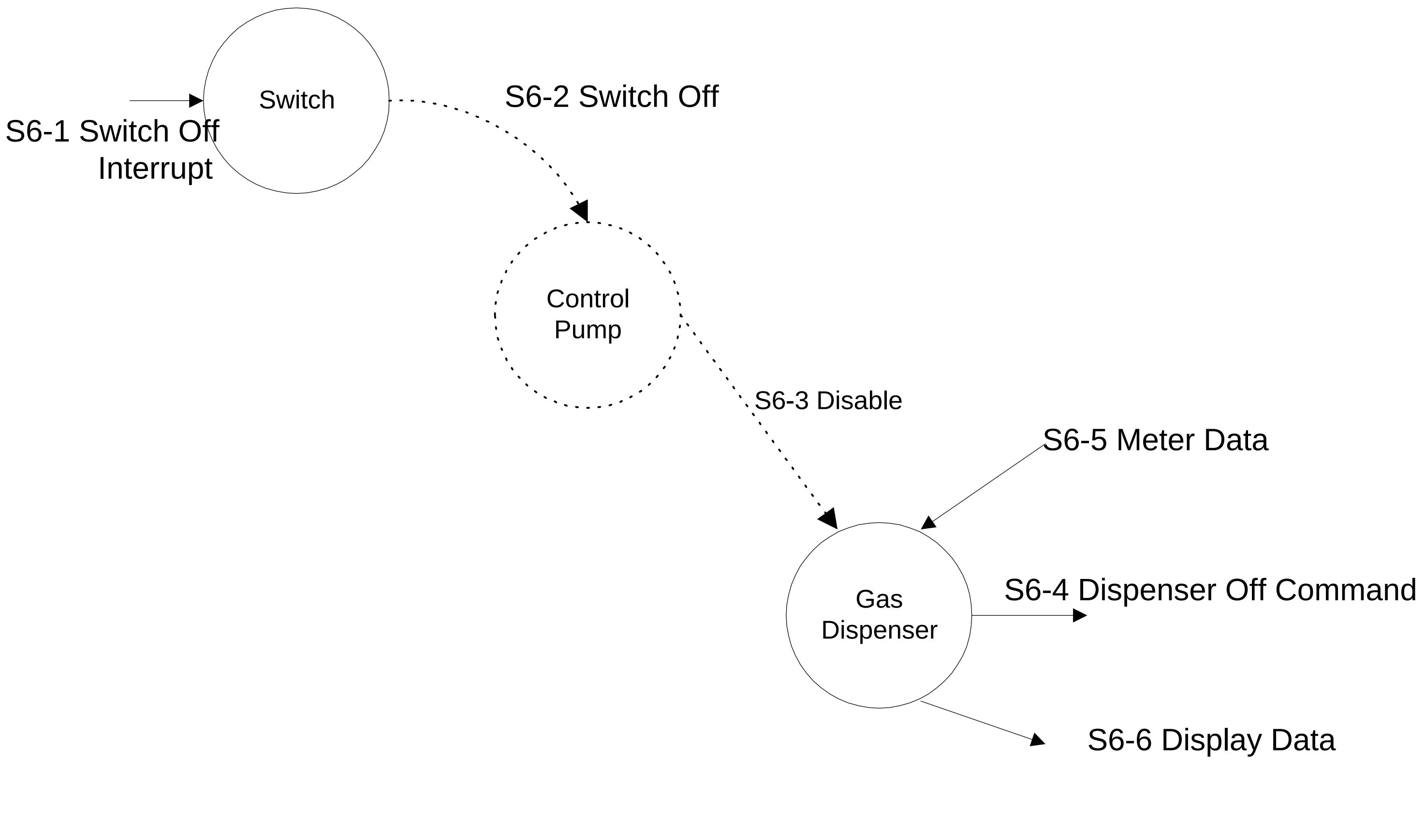
C-34 Scenario #3 - Corporate Credit Card Inserted By The Customer



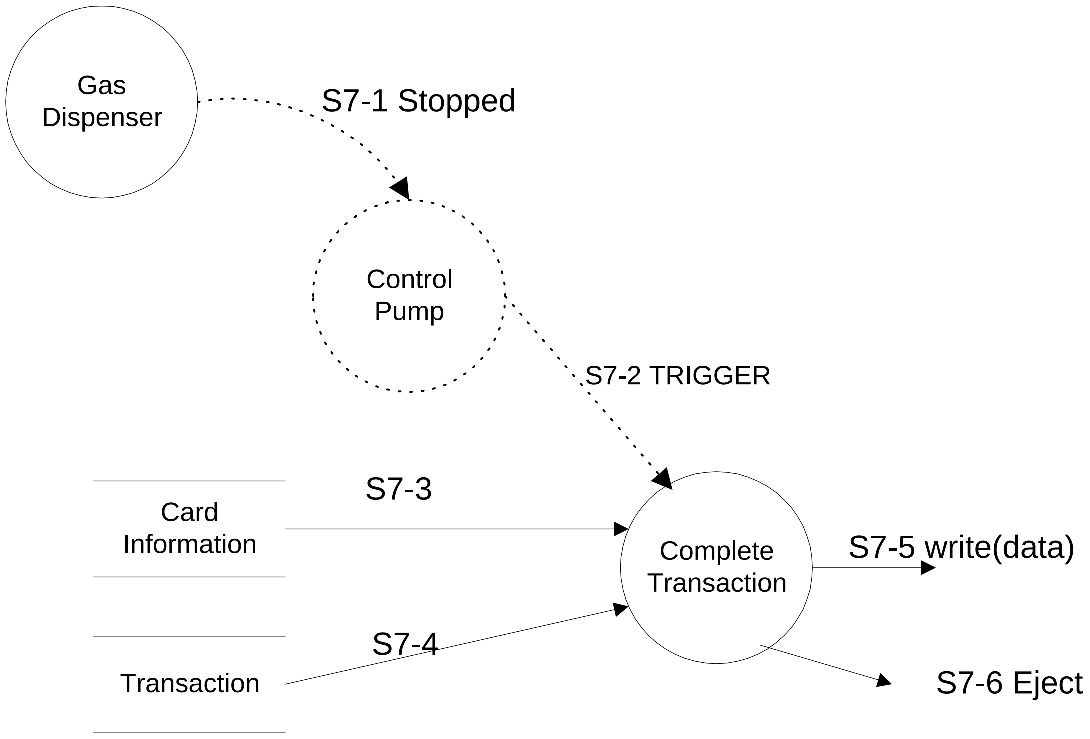
C-35 Scenario #4 - Gas Dispenser Switch Set To On By Customer After The Transaction Has Been Authorized



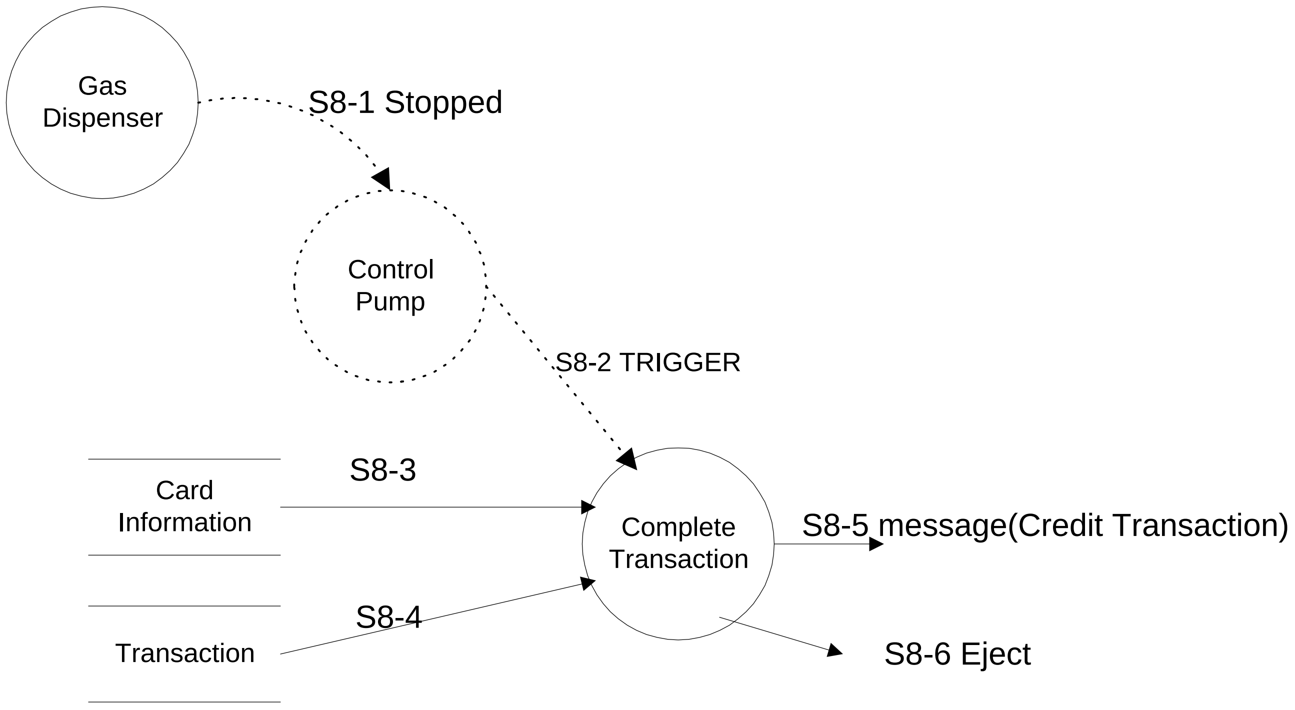
C-36 Scenario #5 - Gas Dispenser Switch Set To On By Customer Before The Transaction Has Been Authorized



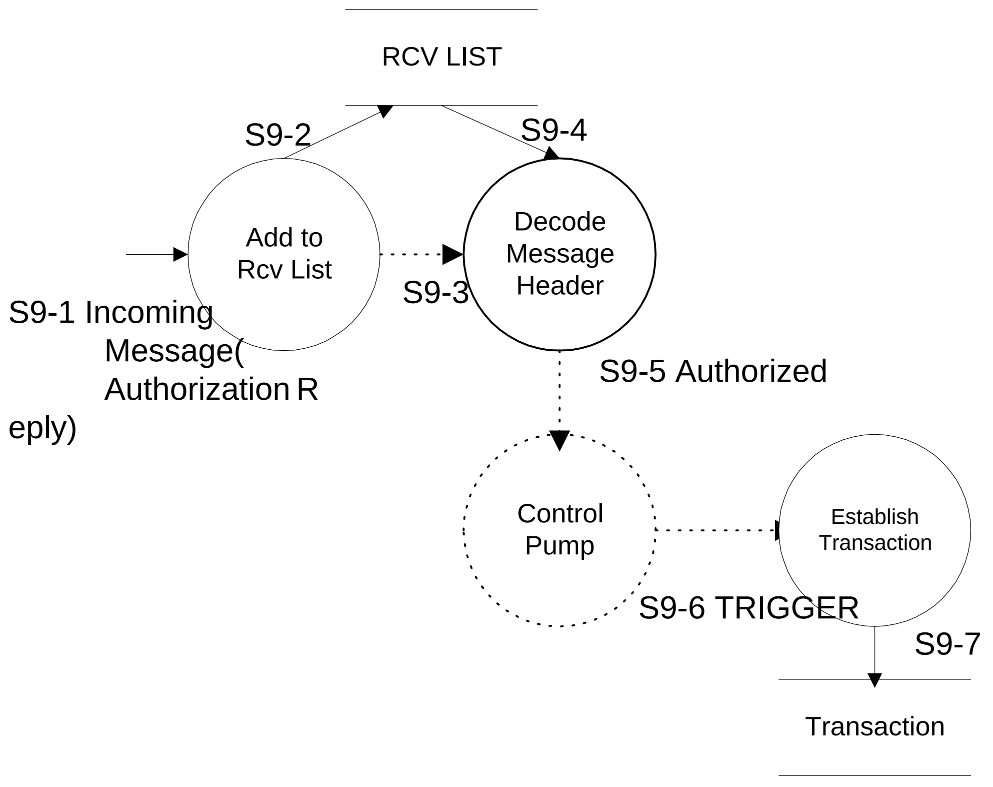
C-37 Scenario #6 - Gas Dispenser Switch Set To Off By Customer



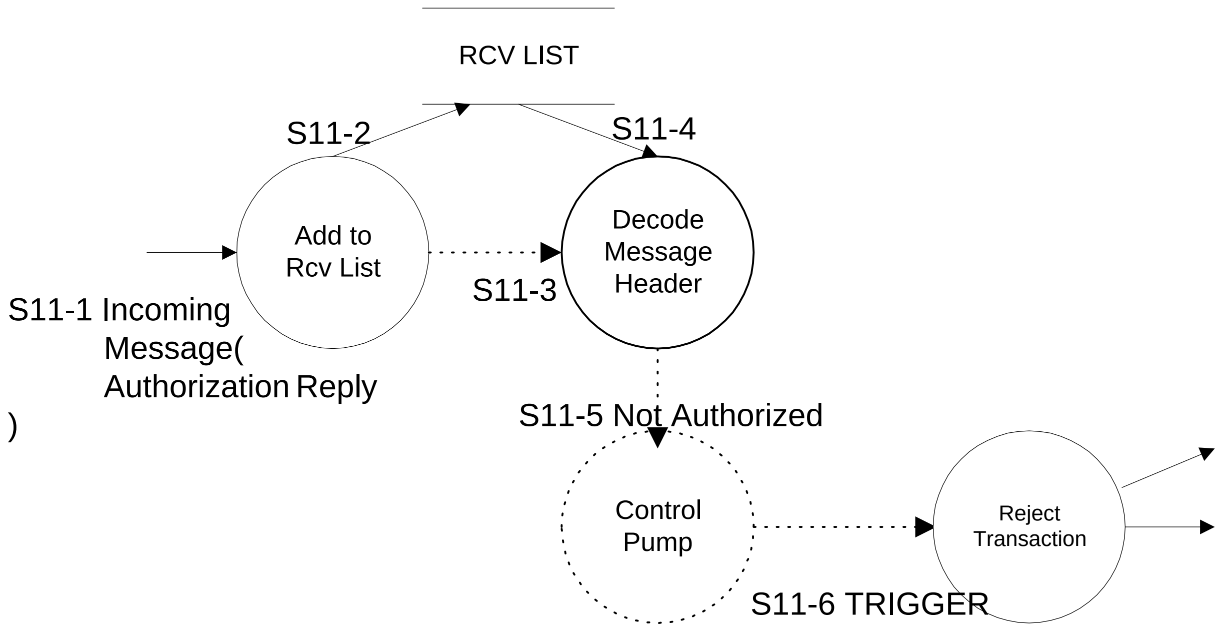
C-38 Scenario #7 - Gas Dispenser Stops For Any Reason (i.e., Disabled or Reached Cash Limit) When A Cash Transaction Was Being Processed



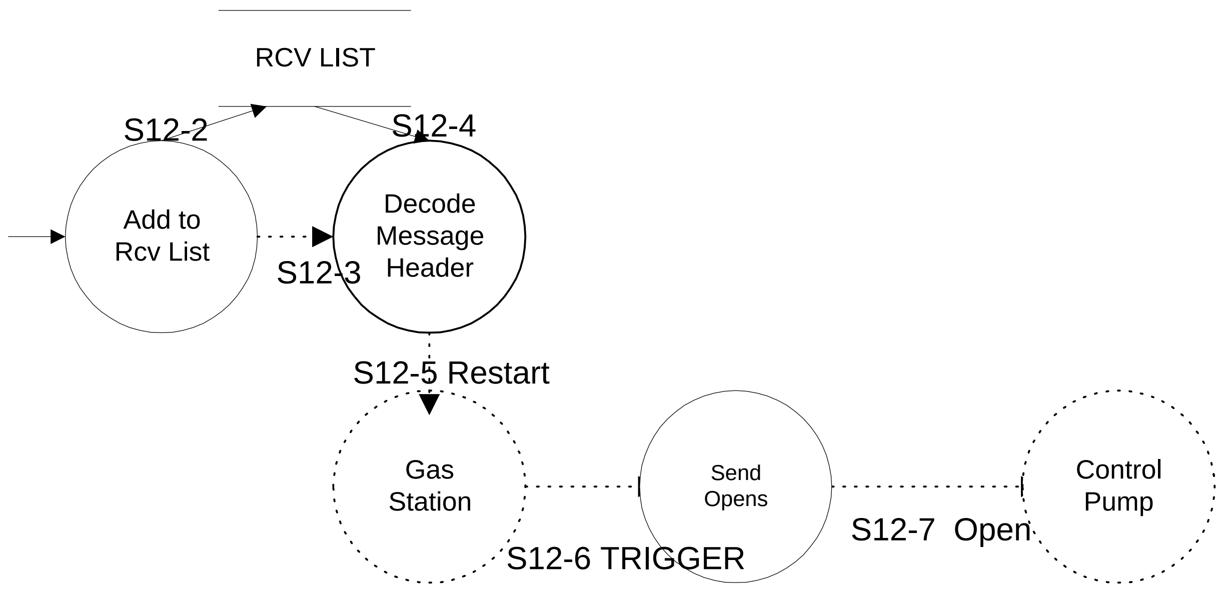
C-39 Scenario #8 - Gas Dispenser Stops When A Credit Transaction Was Being Processed



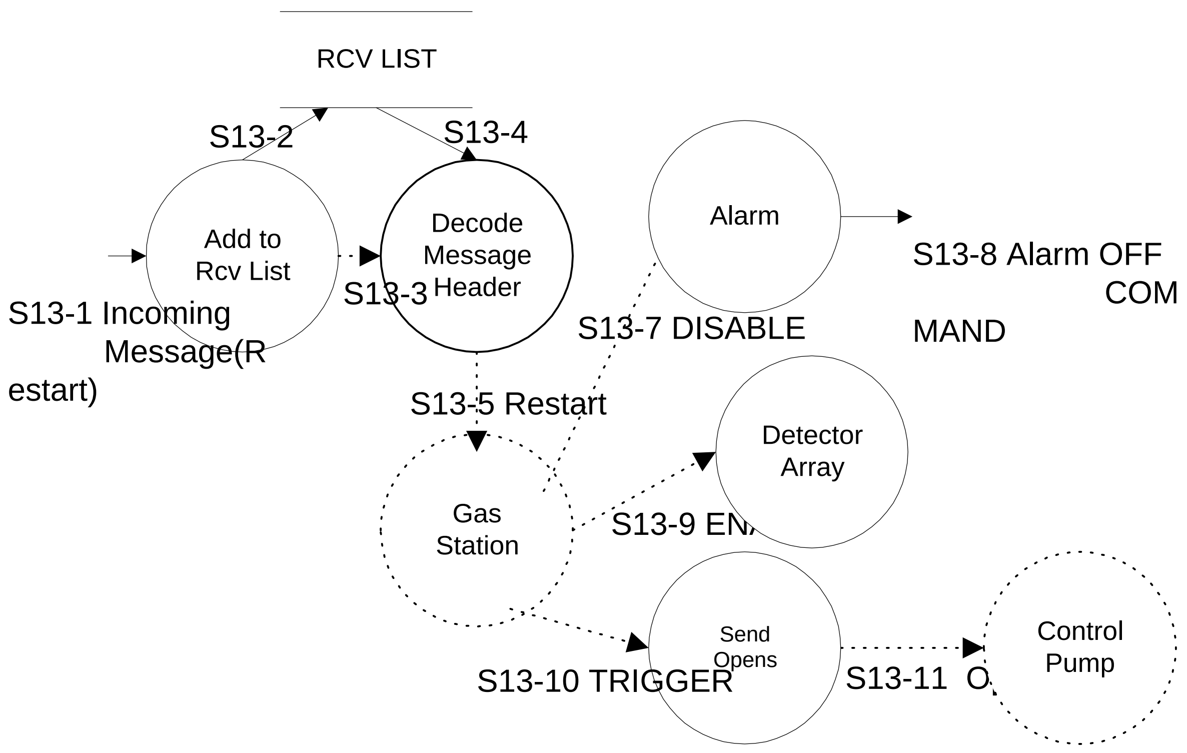
C-40 Scenario #9 - Incoming Authorization Reply With Authorized - Gas Dispenser Switch Is Not On



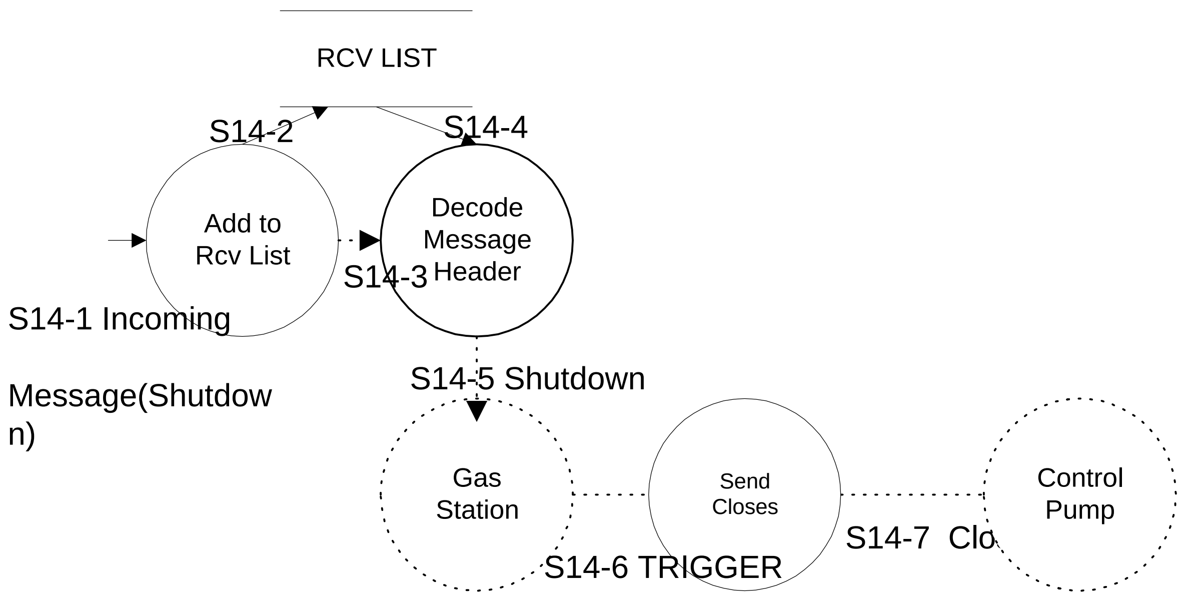
C-42 Scenario #11 - Incoming Authorization Reply With Not Authorized



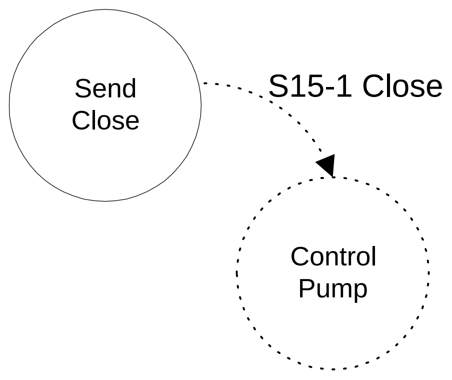
C-43 Scenario #12 - Incoming Restart When Station Was Previously Shutdown By Remote Central Facility



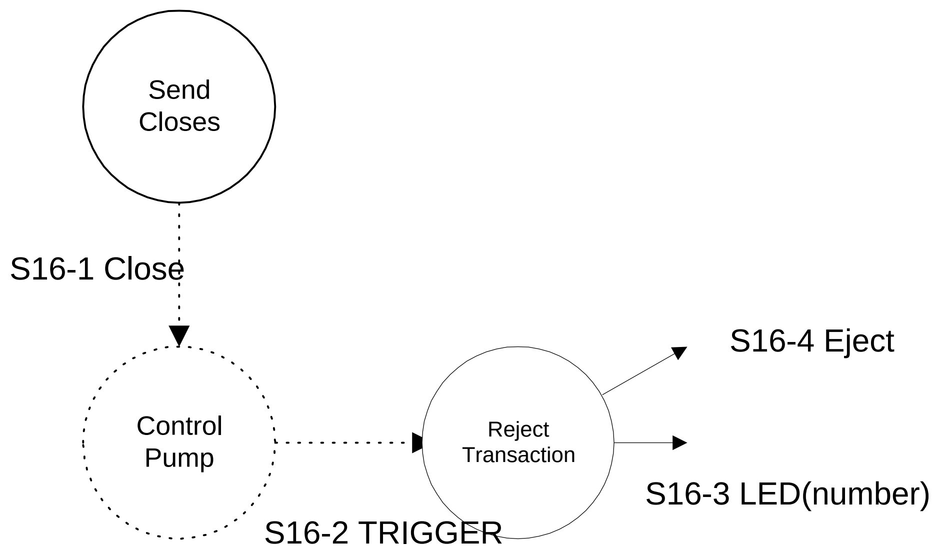
C-44 Scenario #13 - Incoming Restart When Station Was Previously Shutdown By Emergency



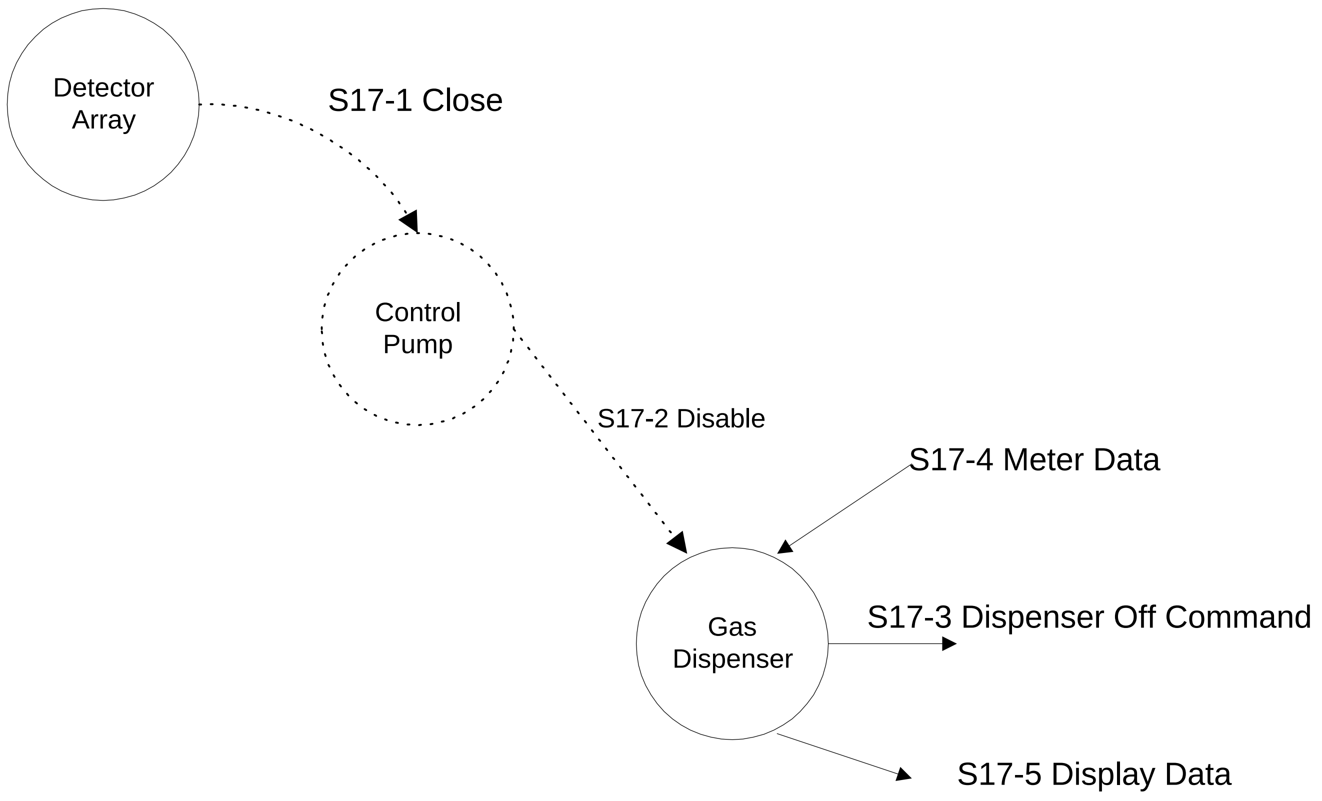
C-45 Scenario #14 - Incoming Shutdown From Remote Central Facility



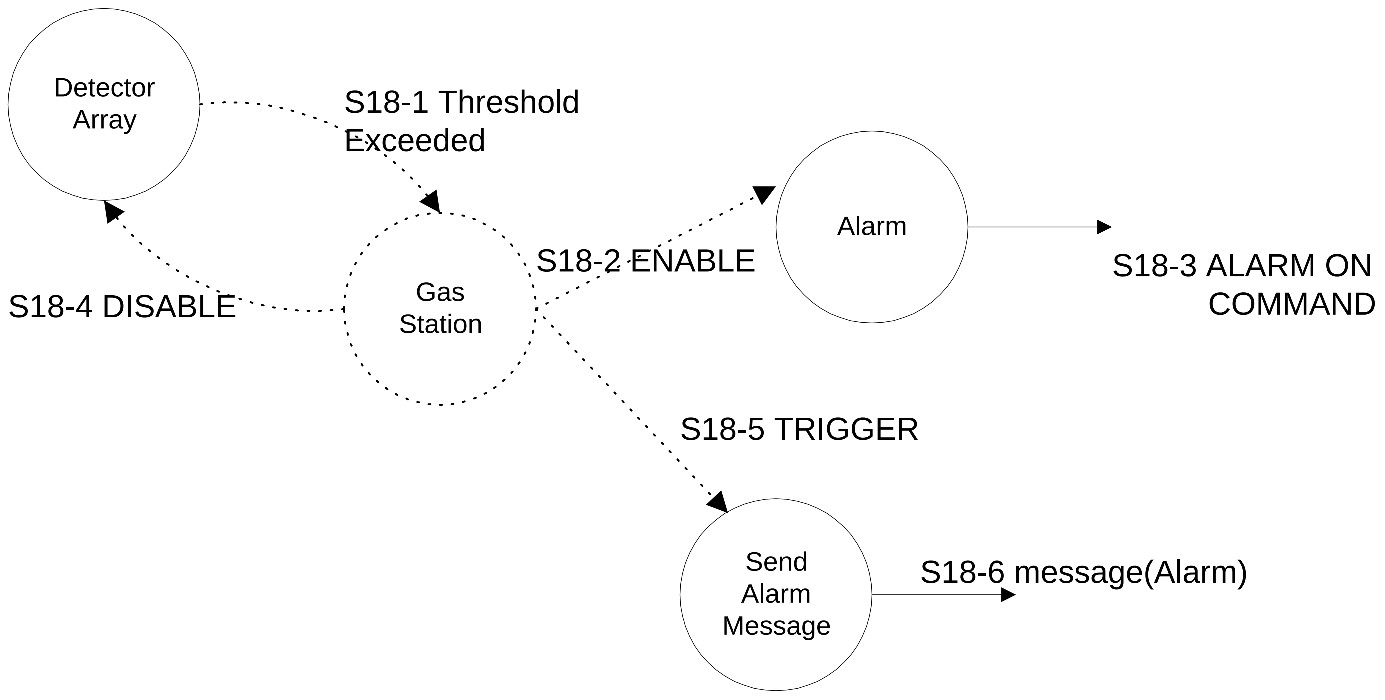
C-46 Scenario #15 - Pump Receives A Close When The Pump Is Idle



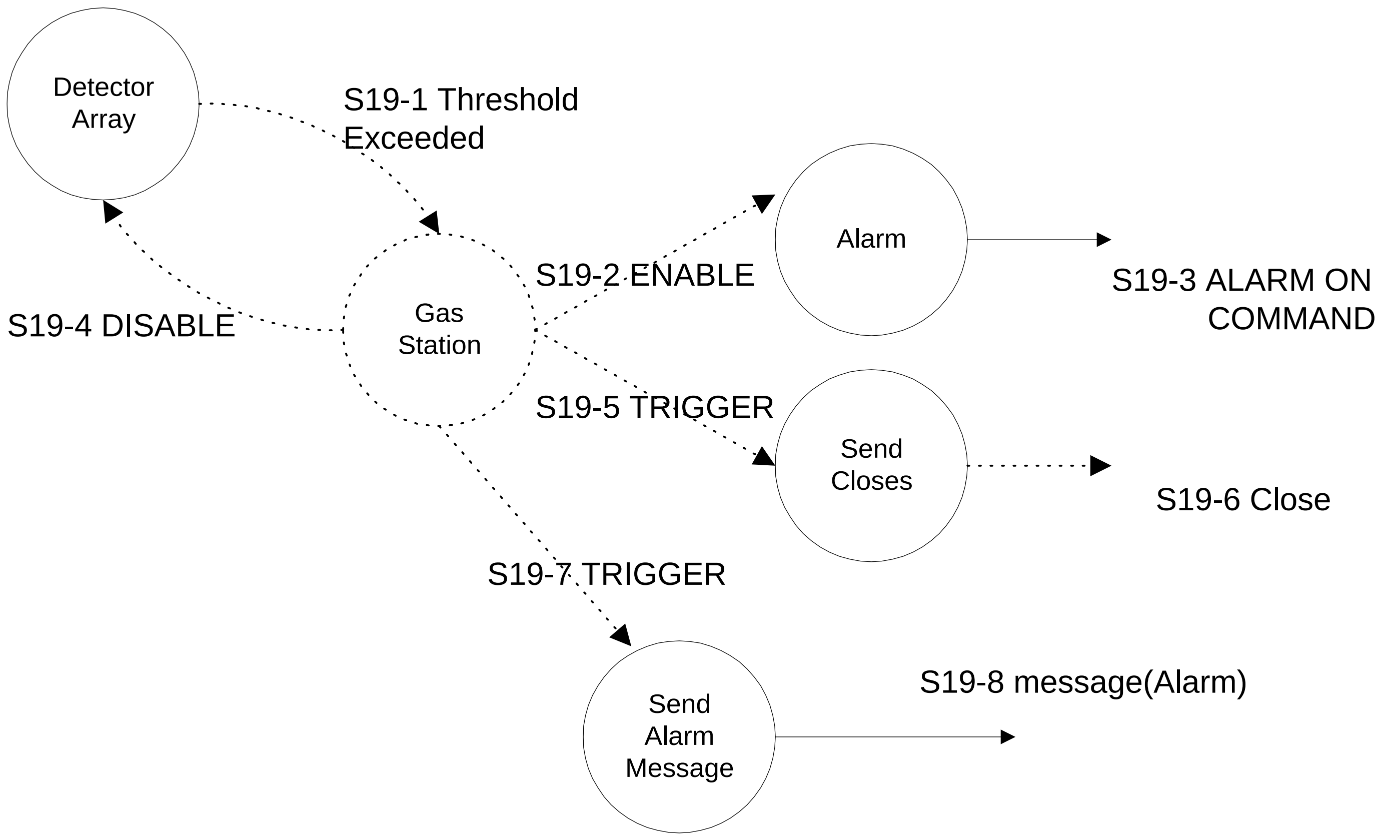
C-47 Scenario #16 - Pump Receives Close While Permission to Dispense Gas Is Pending



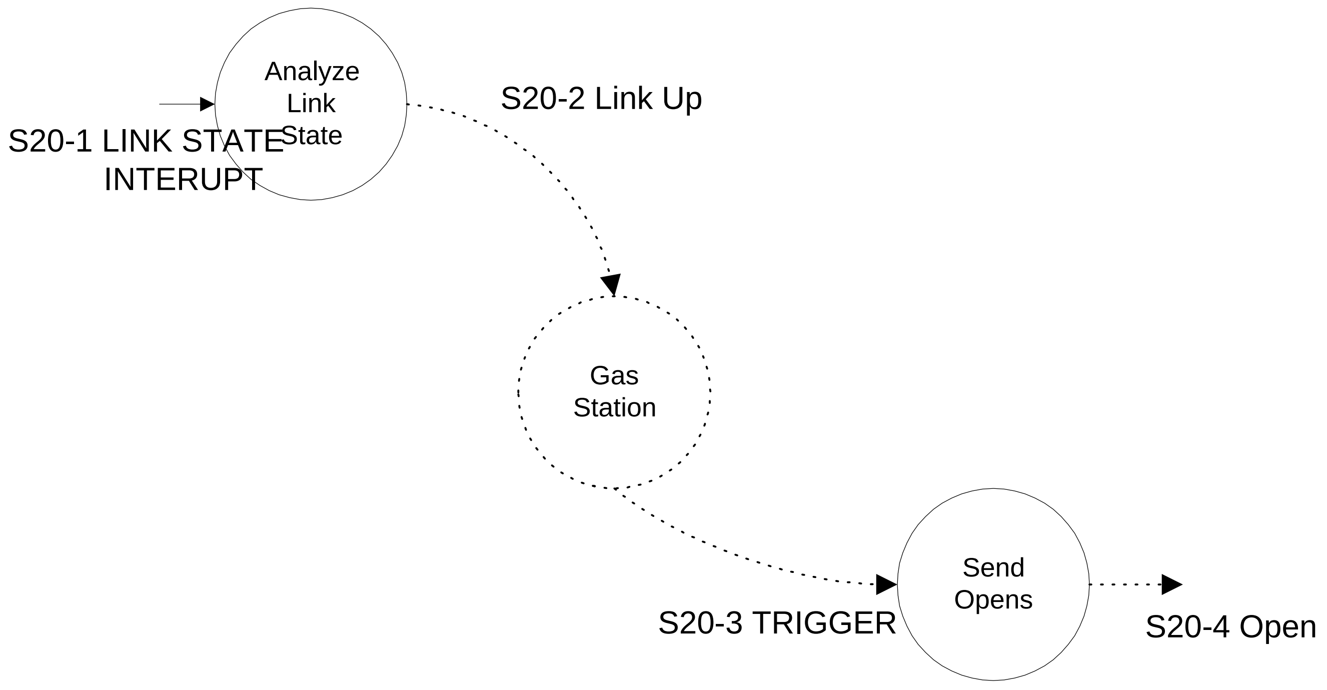
C-48 Scenario #17 - Pump Receives Close While Dispensing Gas



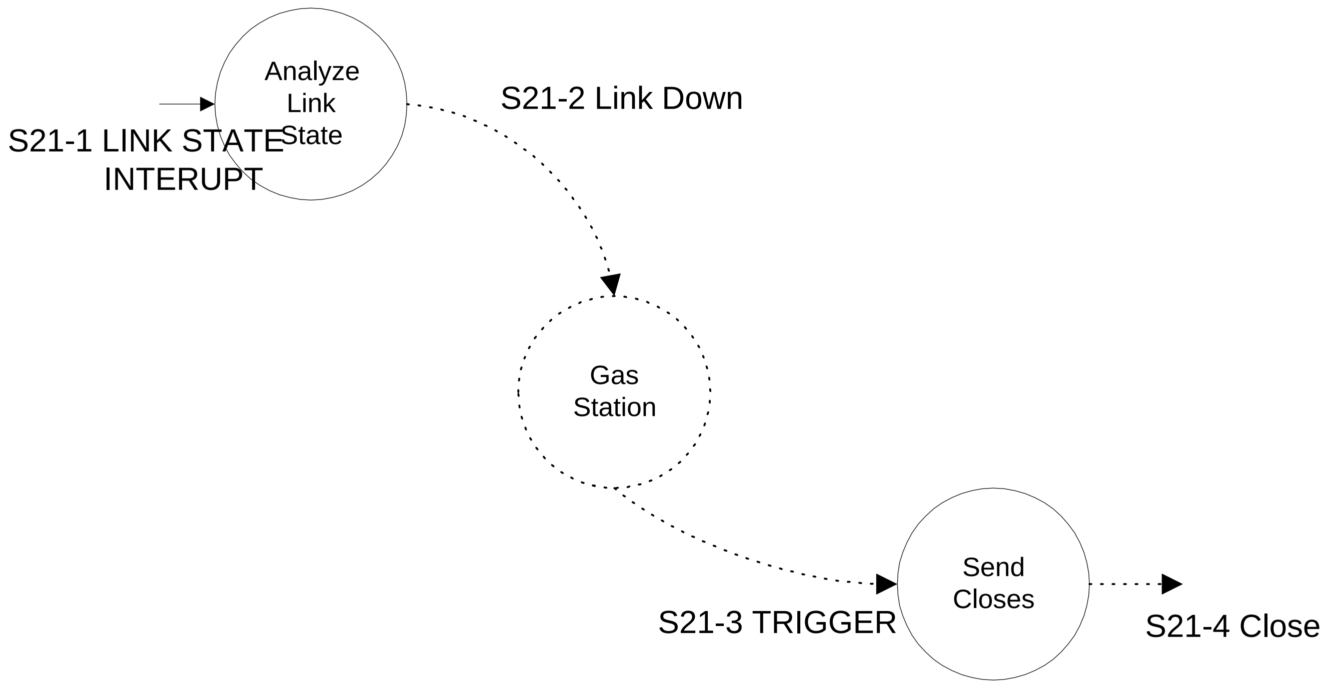
C-49 Scenario #18 - Fire Detected When The Gas Station Is Not Operating



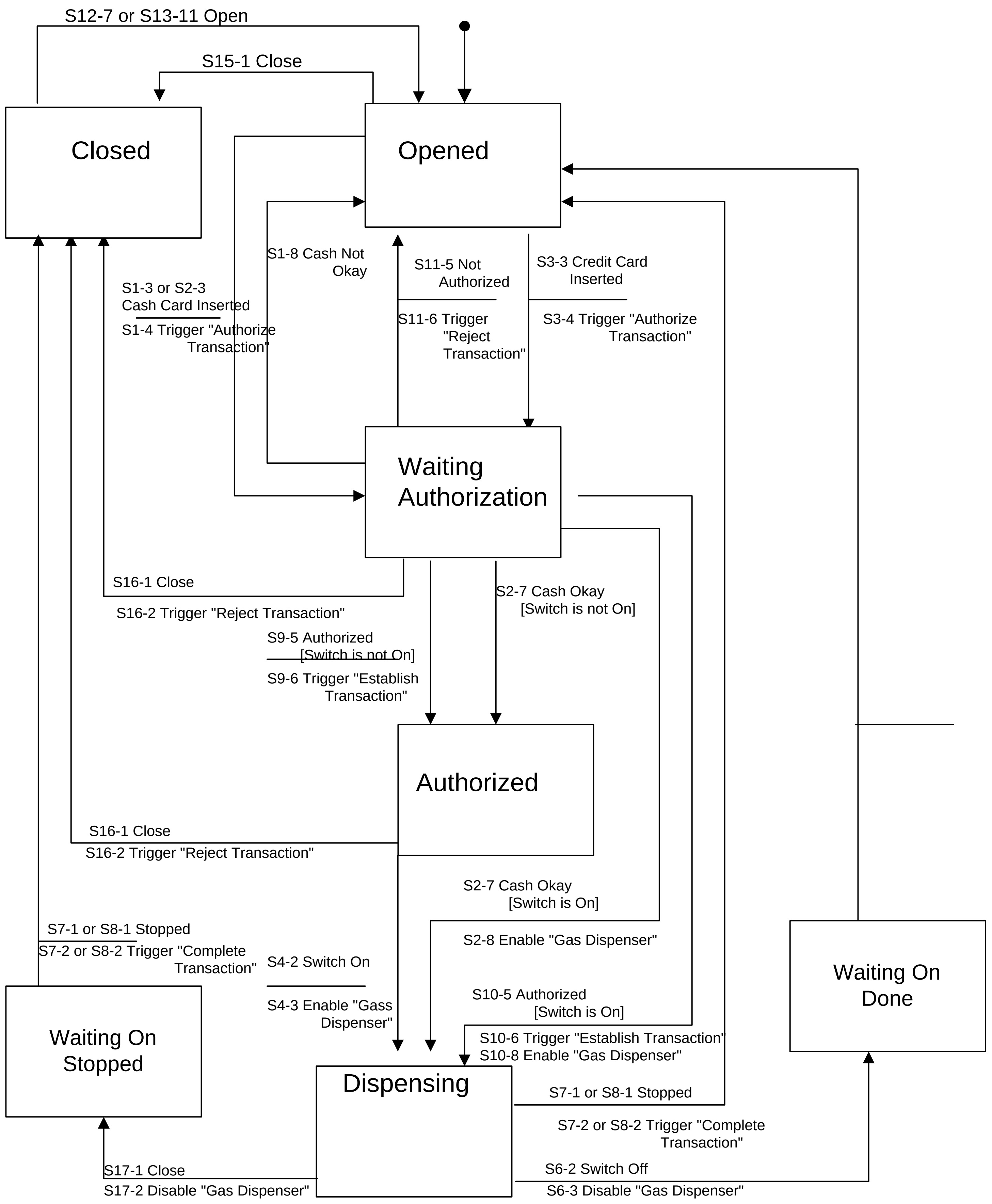
C-50 Scenario #19 - Fire Detected When The Gas Station Is Operating



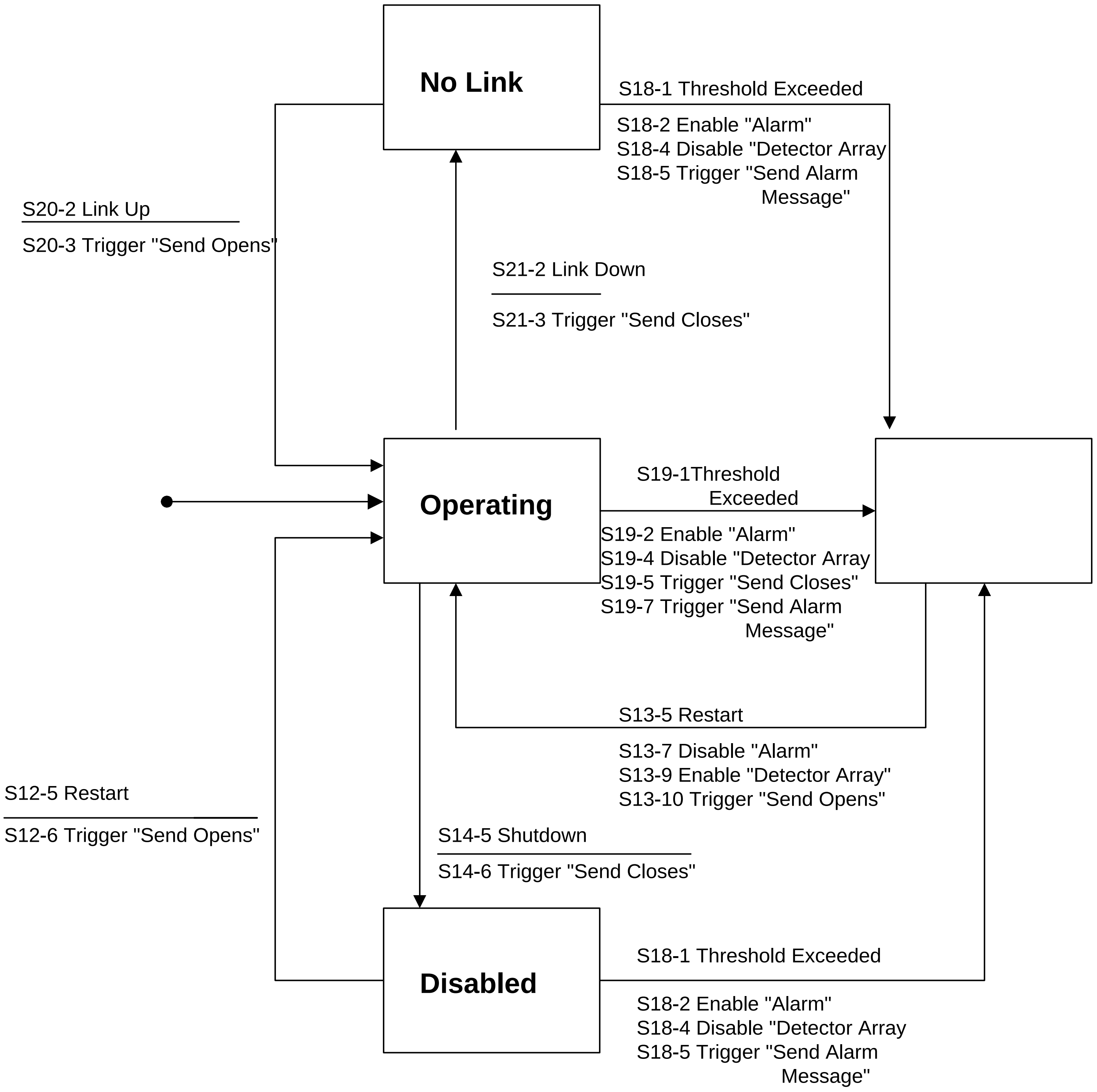
C-51 Scenario #20 Link State Interrupt Brings The Communications Link Up



C-52 Scenario #21 Link State Interrupt Brings The Communications Link Down


S12-7 or S13-11 Open
S15-1 Close
Closed
Opened
S1-8 Cash Not Okay
S11-5 Not Authorized
S11-6 Trigger "Reject Transaction"
S3-3 Credit Card Inserted
S3-4 Trigger "Authorize Transaction"
S1-3 or S2-3 Cash Card Inserted
S1-4 Trigger "Authorize Transaction"
Waiting Authorization
S16-1 Close
S16-2 Trigger "Reject Transaction"
S2-7 Cash Okay [Switch is not On]
S9-5 Authorized [Switch is not On]
S9-6 Trigger "Establish Transaction"
Authorized
S16-1 Close
S16-2 Trigger "Reject Transaction"
S2-7 Cash Okay [Switch is On]
S2-8 Enable "Gas Dispenser"
S7-1 or S8-1 Stopped
S7-2 or S8-2 Trigger "Complete Transaction"
S4-2 Switch On
S4-3 Enable "Gass Dispenser"
S10-5 Authorized [Switch is On]
S10-6 Trigger "Establish Transaction"
S10-8 Enable "Gas Dispenser"
Waiting On Stopped
Dispensing
Waiting On Done
S7-1 or S8-1 Stopped
S7-2 or S8-2 Trigger "Complete Transaction"
S17-1 Close
S17-2 Disable "Gas Dispenser"
S6-2 Switch Off
S6-3 Disable "Gas Dispenser"

C-53 CONTROL PUMP STATE TRANSITION DIAGRAM


No Link
Operating
Disabled
S18-1 Threshold Exceeded
S18-2 Enable "Alarm"
S18-4 Disable "Detector Array
S18-5 Trigger "Send Alarm Message"
S20-2 Link Up
S20-3 Trigger "Send Opens"
S21-2 Link Down
S21-3 Trigger "Send Closes"
S19-1Threshold Exceeded
S19-2 Enable "Alarm"
S19-4 Disable "Detector Array
S19-5 Trigger "Send Closes"
S19-7 Trigger "Send Alarm Message"
S13-5 Restart
S13-7 Disable "Alarm"
S13-9 Enable "Detector Array"
S13-10 Trigger "Send Opens"
S12-5 Restart
S12-6 Trigger "Send Opens"
S14-5 Shutdown
S14-6 Trigger "Send Closes"
S18-1 Threshold Exceeded
S18-2 Enable "Alarm"
S18-4 Disable "Detector Array
S18-5 Trigger "Send Alarm Message"

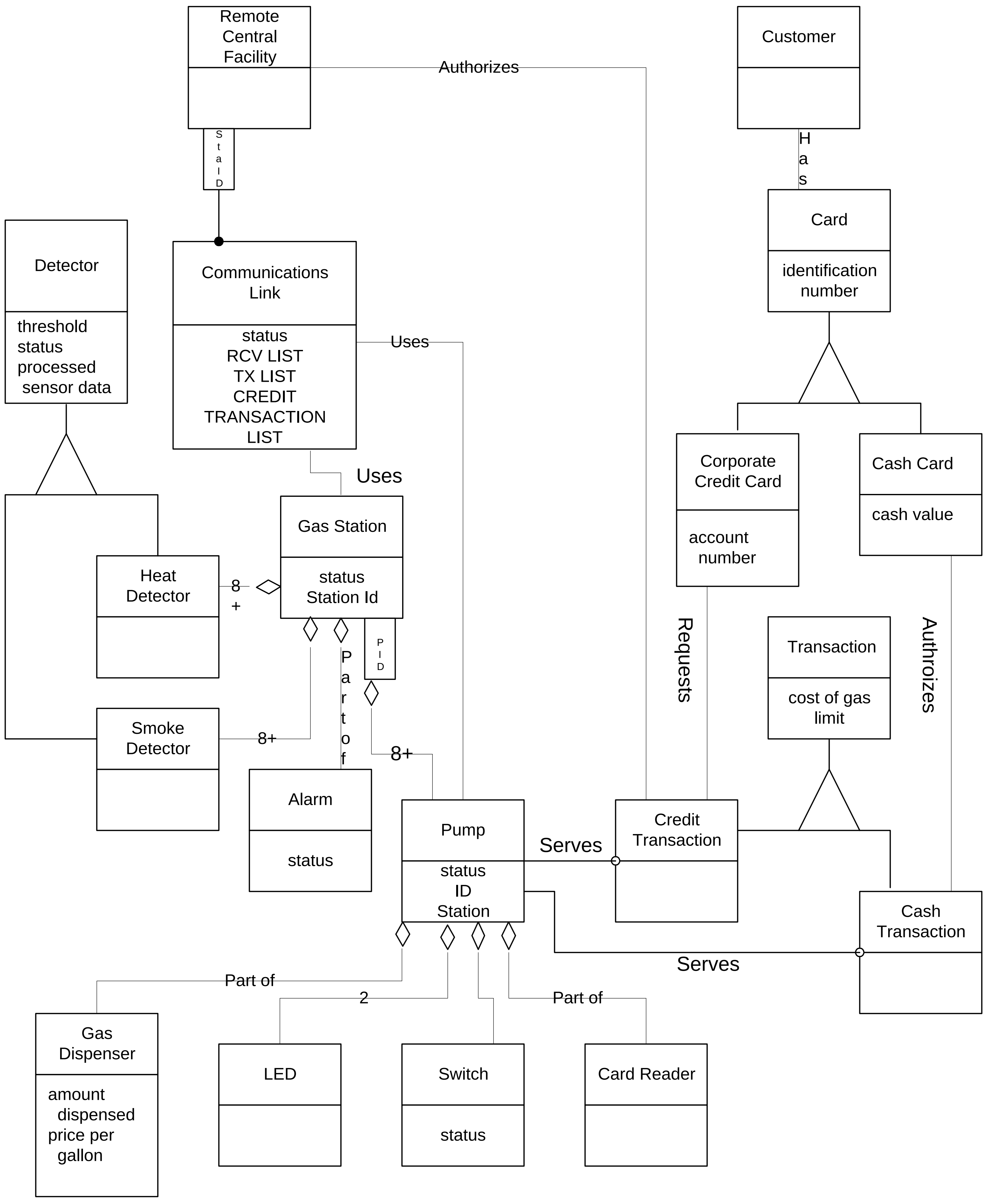
C-54 Gas Station STATE TRANSITION DIAGRAM

APPENDIX D. AUTOMATED GAS STATION MANAGER OMT SPECIFICATION

AUTOMATED GAS STATION MANAGER

OBJECT MODEL

AGSM OBJECT MODEL



Object Dictionary For Automated Gas Station Manager (AGSM)

Alarm

A audible device that is sounded at a ***Gas Station*** whenever a fire is detected at the station.

Attributes

1. status: BOOLEAN

Operations

1. sound -- Turns on alarm
2. reset -- Turns off alarm

Card

A token, possessed by a ***Customer***, that can be inserted into a ***Card Reader***. Each ***Card*** has an identification number to allow the ***Card Reader*** to distinguish various types of ***Cards***.

Attributes

1. identification number: NUMERIC

Operations

1. set_id_number(id: NUMERIC)
2. get_id_number:NUMERIC
3. set_account_number(x:NUMERIC) -- default
4. get_account_number:NUMERIC -- default
5. set_cash_value(x:INTEGER) -- default
6. get_cash_value:INTEGER -- default

Card Reader

An input/output device attached to each ***Pump*** that can detect the insertion of a ***Corporate Credit Card*** or a ***Cash Card***, that can distinguish between the two types of cards (as well as identify unrecognized cards), that can read appropriate information from each type of card, that can debit and write a new balance to a ***Cash Card***, and that can eject a card.

Operations

1. write(id_num: NUMERIC, value: INTEGER)
2. Eject
3. read

Cash Card inherits Card

A card encoded with a cash value which is debited by the dollar amount of gas pumped after each transaction. The cash balance is printed on the card after each transaction. The card contains an identification number which distinguishes it from a ***Corporate Credit Card*** and from other types of cards that are not known to the system.

<u>Attributes</u>

1. cash value: INTEGER

<u>Operations</u>

1. set_cash_value(amount: INTEGER)
2. get_cash_value: INTEGER

Cash Transaction inherits Transaction

A transaction authorized by the cash value contained on a ***Cash Card***. Gas may be dispensed up to the amount of the cash value.

<u>Operations</u>

1. complete(card:CASH CARD) -- completes processing a Cash Transaction
2. authorize(card:CASH CARD) -- determines whether a Cash Card is valid
3. reject -- lights appropriate LED and Ejects Card

Corporate Credit Card inherits Card

A card encoded with a corporate account number which must be verified before dispensing gas. The card also contains an identification number which distinguishes it from a ***Cash Card*** and from other types of cards that are not known to the system.

<u>Attributes</u>

1. account number: NUMERIC

<u>Operations</u>

1. get_account_number: NUMERIC
2. set_account_number(account: NUMERIC)

Credit Transaction inherits Transaction

A transaction requested with a ***Corporate Credit Card*** and authorized by the ***Remote Central Facility***. Once authorized, gas may be dispensed without limit.

<u>Operations</u>

1. complete(card:CREDIT CARD) -- completes processing a credit transaction
2. authorize(card:CREDIT CARD) -- determines whether a Credit Card is valid

3. reject -- lights appropriate LED and Ejects Card

Communications Link

A link between the ***Gas Station*** and the ***Remote Central Facility***. The link can go up and down. When the link is up, Validation Requests and Validation Replies may be exchanged between ***Pumps*** and the ***Remote Central Facility*** and Alarm Messages and ***Credit Transactions*** may be sent from the ***Gas Station*** to the ***Remote Central Facility***. The ***Gas Station*** can also receive Shutdown Commands from the ***Remote Central Facility***. When the link goes down, the ***Gas Station*** shuts down, except that any pending ***Credit Transactions*** are saved on secondary storage so that the processing can be completed when the link comes back up.

Attributes

1. status: INTEGER
2. RCV_LIST: LINKED_LIST
3. TX_LIST: LINKED_LIST
4. CREDIT_TRANSACTION_LIST: FILE

Operations

1. analyze_link(event: LINK_EVENT) -- encapsulates Link STD
2. add_to_rcv_list(message: ARRAY[BYTES]) -- enqueues a received message
3. add_to_tx_list(message: RECORD) -- enqueues a message for transmission
4. decode_message_header -- generates incoming events for other STDs
5. transmit_message -- sends a message if possible
6. handle_message_sent_interrupt
7. handle_message_received_interrupt
8. handle_link_state_interrupt
9. save_credit_transactions   -- moves credit transactions from TX_LIST to
   -- CREDIT_TRANSACTION_LIST. Other messages are
   -- discarded
10. restore_credit_transactions -- moves credit transactions from
    -- CREDIT_TRANSACTION_LIST to head of TX_LIST

Customer

A human being who initiates a gas purchase transaction by inserting a ***Cash Card*** or a ***Corporate Credit Card*** and who turns the ***Pump*** on and off and who dispenses the gas.

Detector

A device that monitors some physical condition and signals when the monitored condition surpasses a specified threshold.

Attributes

1. threshold: REAL
2. status: INTEGER
3. processed_sensor_data: REAL

Operations

1. read {abstract} -- must read the sensor, convert the data to processed form and store it in
   -- processed_sensor_data
2. monitor_sensor:REAL --computes a delta between threshold and processed sensor data
3. send_threshold_exceeded -- generates a Threshold Exceeded event for the Gas Station
4. process_change(delta:REAL) --encapsulates the Detector STD

Gas Dispenser

The mechanism within each ***Pump*** that dispenses gasoline, measures the amount dispensed, and terminates dispensing when conditions require.

Attributes

1. amount dispensed: REAL
2. price per gallon: INTEGER is CONSTANT

Operations

1. start_gas -- starts the gas dispenser to dispense up to the
   -- maximum, but a zero maximum means no limit
2. stop_gas -- stops the gas dispenser
3. clear_amount_dispensed
4. update_amount_dispensed
5. update_display
6. get_amount_dispensed: REAL
7. get_price: INTEGER
8. ?limit_reached(t:TRANSACTION): BOOLEAN -- T if the limit is reached, F otherwise

Gas Station

A physical location comprising ***Pumps***, an ***Alarm***, a ***Dedicated Communications Link***, and some local, secondary storage. The ***Gas Station*** can monitor and control its ***Pumps***, control its ***Alarm***, and monitor its ***Dedicated Communications Link***.

Attributes

1. status: INTEGER
2. Pump_Ids: LINKED_LIST
3. Station_Id: INTEGER

Operations

1. process_event(event: GS_EVENT)
2. open_pumps -- send an Open Event to each Pump STD
3. close_pumps -- send a Close Event to each Pump STD
4. send_alarm_message -- forward alarm message to the Remote Central Facility

Heat Detector inherits Detector

A device attached to each ***Pump***. The device monitors heat levels and signals the ***Gas Station*** when heat surpasses a specified threshold.

Operations

1. read -- inputs sensor data, converts to REAL, and stores as processed sensor data

LED

A light-emitting diode. Two are attached to each ***Pump***: one is placed under a label "Cannot Process Card" and the other is placed under a label "Card Value Used". The first is lighted when a ***Customer*** inserts a card that is not a valid ***Corporate Credit Card*** or a ***Cash Card***. The second is lighted when the value on an inserted ***Cash Card*** reaches zero. Any ***LED*** that is lit will be turned off when another card is inserted into the ***Pump*** or after a time-out.

Operations

1. light   -- turns itself on and, after a timeout, turns itself off.

Pump

A gasoline dispensing device located at a ***Gas Station***. Each ***Pump*** can dispense one grade of gasoline. Each ***Pump*** comprises a ***Heat Detector***, a ***Smoke Detector***, two ***LED***s, a ***Card Reader***, a ***"Gas Purchased" Display***, a ***"Gas Cost" Display***, an on/off switch, and a nozzle.

Attributes

1. status: INTEGER
2. ID: INTEGER -- identity of the Pump
3. Station: INTEGER -- identity of the station where the pump is located

Operations

1. process_event(event: PUMP_EVENT) -- encapsulates the Pump STD

Remote Central Facility

A control point for a ***Gas Station***. The ***Remote Central Facility*** communicates with a ***Gas Station*** across a ***Communications Link***. The ***Remote Central Facility*** accepts ***Credit Transactions*** from ***Pumps***, Validation Requests, and Alarm Messages. The ***Remote Central Facility*** issues Shutdown Commands, and Validation Replies**.**

Smoke Detector inherits Detector

A device attached to each ***Pump***. The device monitors concentration of smoke particles and signals the ***Gas Station*** when the concentration of smoke particles surpasses a specified threshold.

<u>Operations</u>

1. read -- inputs sensor data, converts to REAL, and stores as processed sensor data

Switch

The on/off actuator at each ***Pump***. The ***Customer*** must turn the ***Switch*** on before dispensing gasoline and must turn the ***Switch*** off when finished dispensing gasoline.

<u>Attributes</u>

1. status: INTEGER

<u>Operations</u>

1. handle_switch_interrupt -- Switch interrupt processing
2. process_event(event: SWITCH_EVENT) -- encapsulates Switch STD
3. send_switch_on -- generate switch on event for Pump
4. send_switch_off -- generate switch off event for Pump

Transaction

A ***Transaction*** is the filling of a ***Customer***'s request for gasoline. Each ***Transaction*** results in a ***Customer*** being charged for the cost of gas purchased.

<u>Attributes</u>

1. cost of gas: INTEGER
2. limit: INTEGER

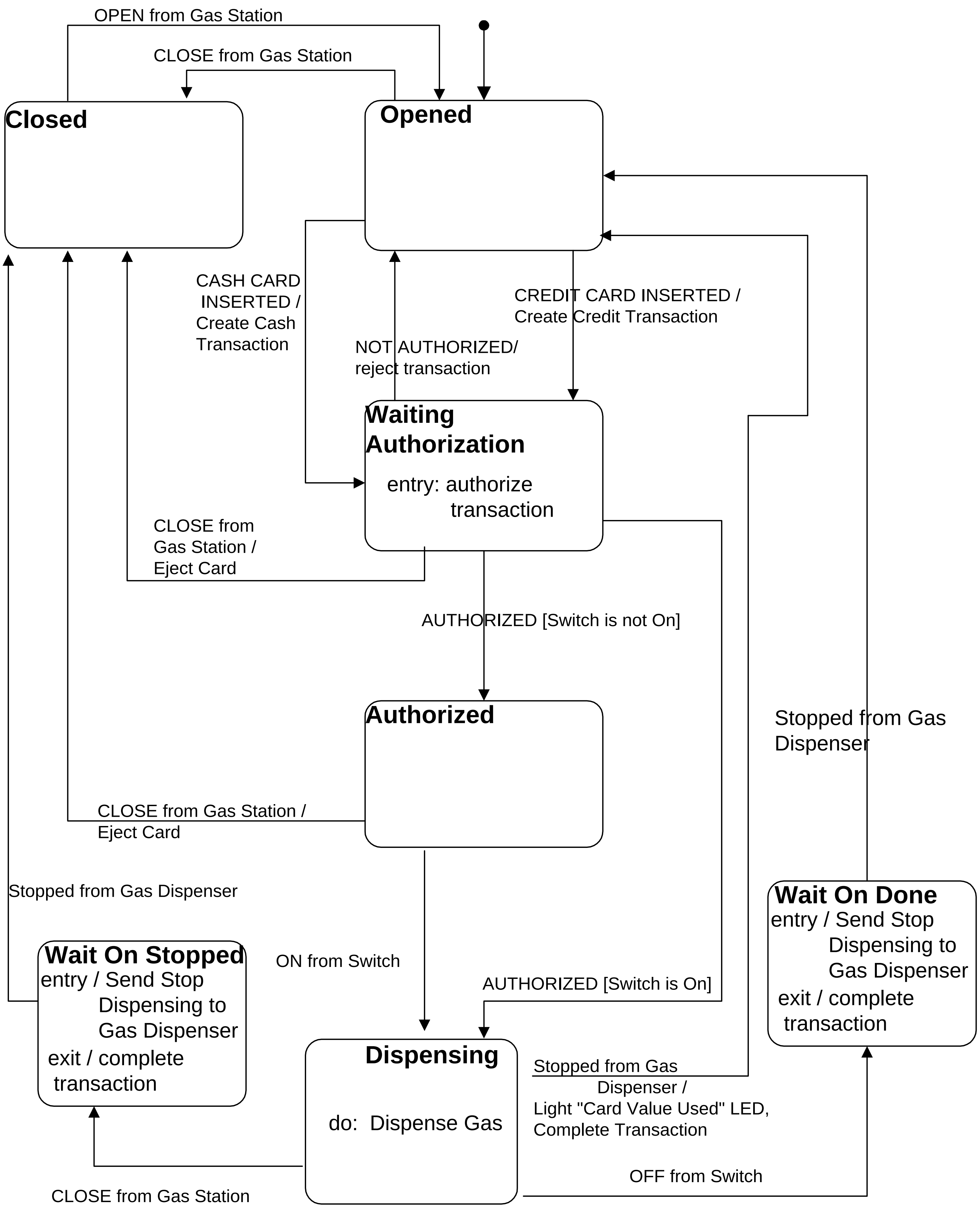
<u>Operations</u>

1. get_limit: INTEGER -- returns the value of the limit attribute
2. set_cost_of_gas(amount: INTEGER) -- changes the value of the cost of gas attribute
3. complete(card:CARD) {abstract} -- complete the transaction

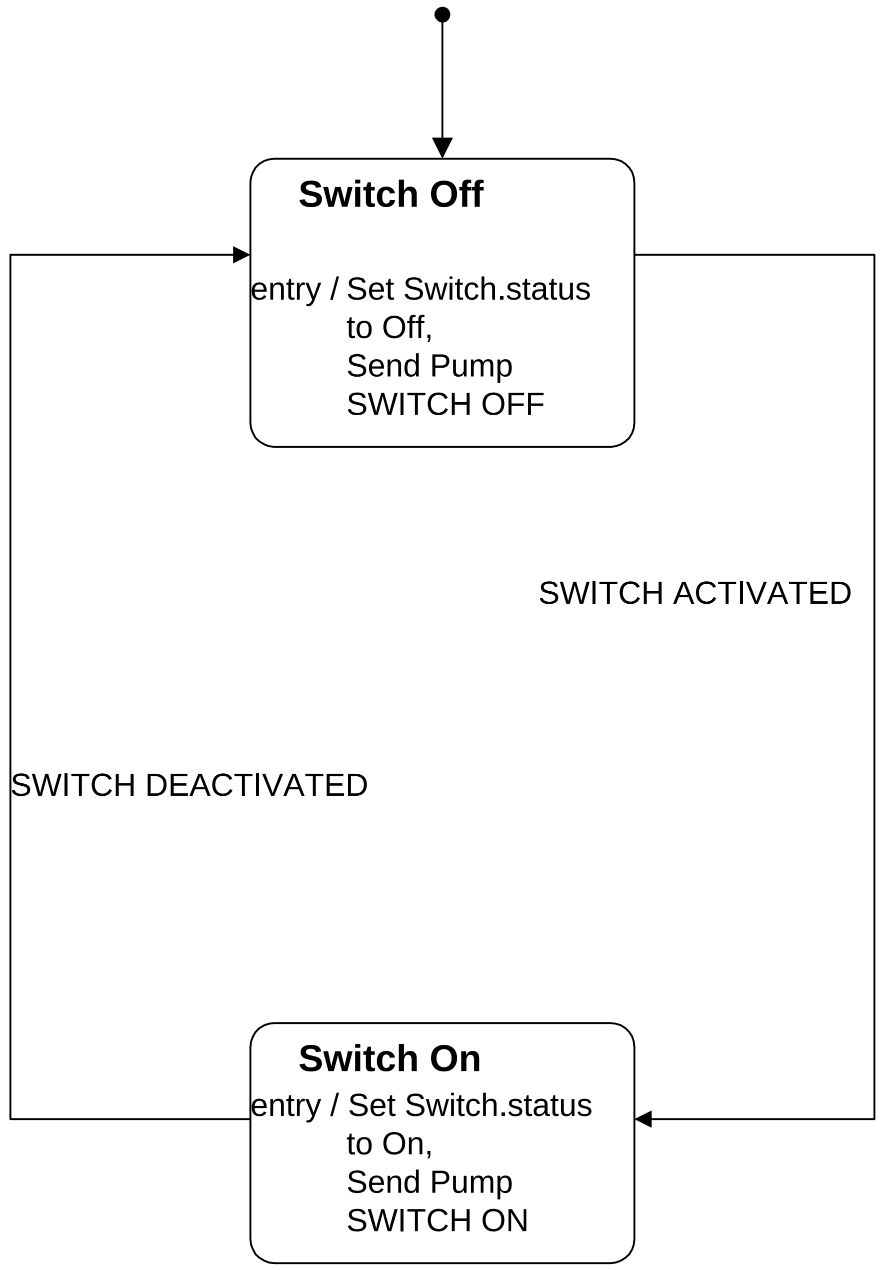
4. authorize(card:CARD) {abstract}   -- decide whether the transaction is authorized
5. reject {abstract}   -- terminate an unauthorized transaction

AUTOMATED GAS STATION MANAGER

DYNAMIC MODEL

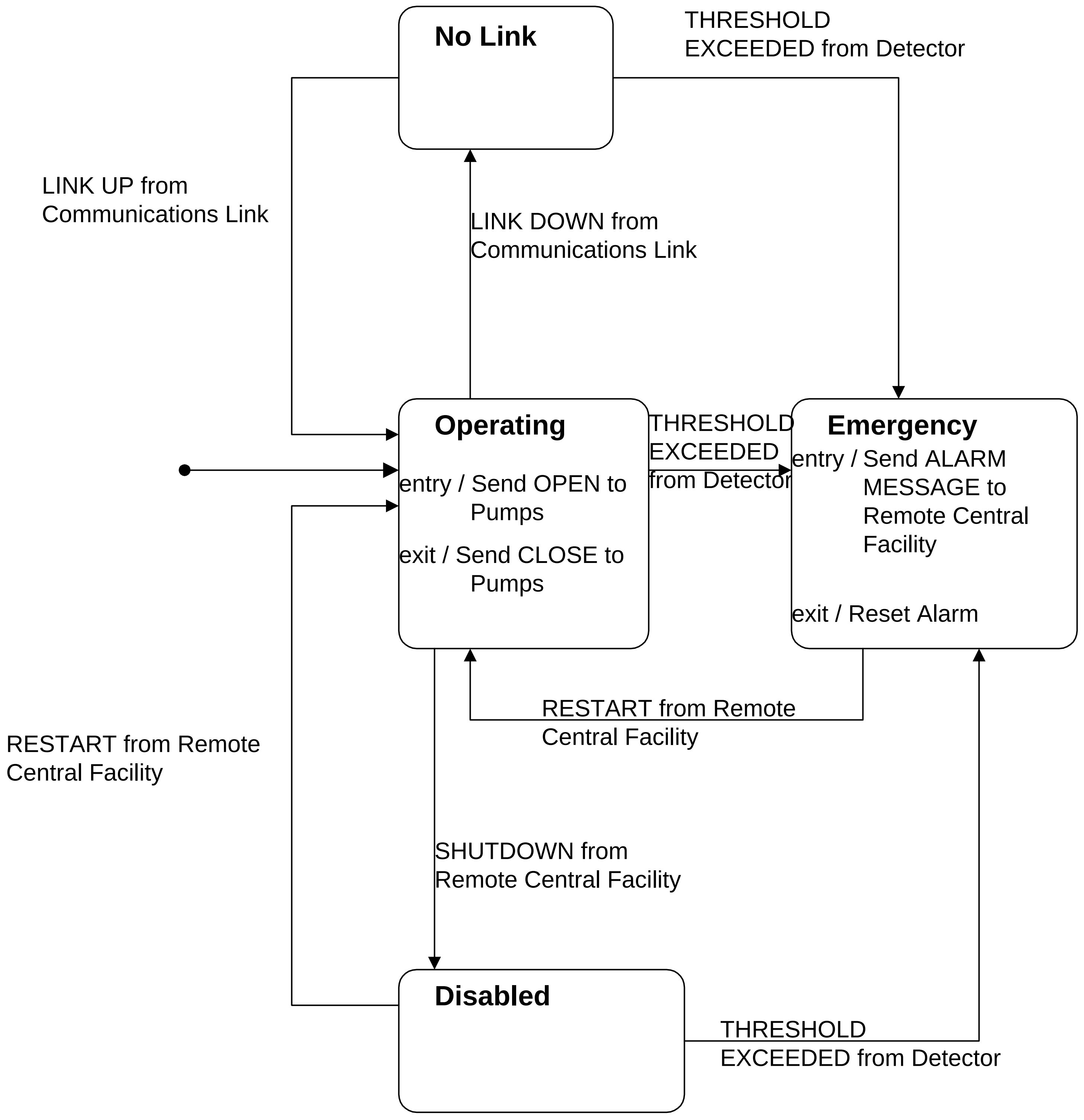

OPEN from Gas Station
CLOSE from Gas Station
Closed
Opened
CASH CARD INSERTED / Create Cash Transaction
CREDIT CARD INSERTED / Create Credit Transaction
NOT AUTHORIZED/ reject transaction
Waiting Authorization
entry: authorize transaction
CLOSE from Gas Station / Eject Card
AUTHORIZED [Switch is not On]
Authorized
Stopped from Gas Dispenser
CLOSE from Gas Station / Eject Card
Stopped from Gas Dispenser
Wait On Done
entry / Send Stop Dispensing to Gas Dispenser
exit / complete transaction
Wait On Stopped
entry / Send Stop Dispensing to Gas Dispenser
exit / complete transaction
ON from Switch
AUTHORIZED [Switch is On]
Dispensing
do: Dispense Gas
Stopped from Gas Dispenser / Light "Card Value Used" LED, Complete Transaction
OFF from Switch
CLOSE from Gas Station

D-11 PUMP STATE TRANSITION DIAGRAM

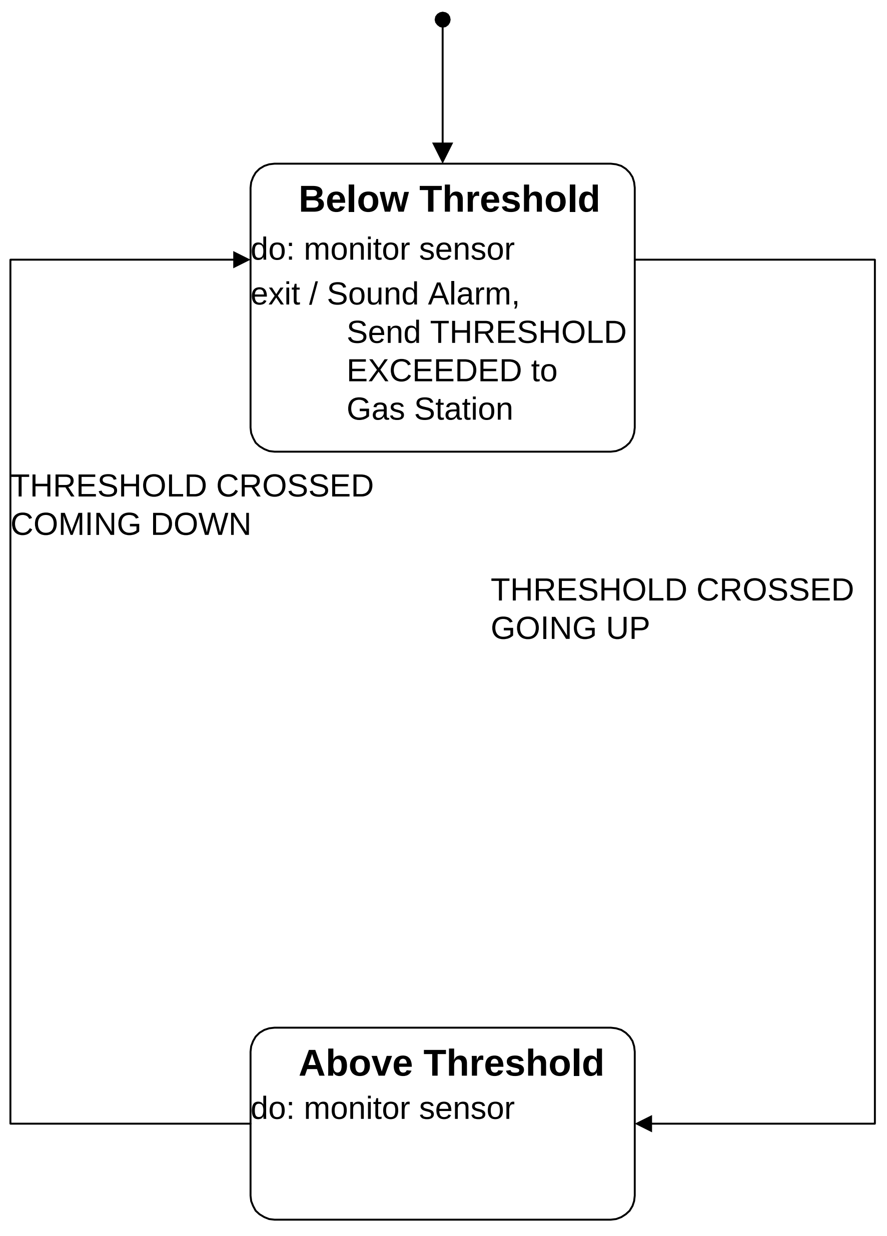


Switch Off
entry / Set Switch.status to Off, Send Pump SWITCH OFF
SWITCH ACTIVATED
SWITCH DEACTIVATED
Switch On
entry / Set Switch.status to On, Send Pump SWITCH ON

D-12 SWITCH STATE TRANSITION DIAGRAM


No Link
THRESHOLD EXCEEDED from Detector
LINK UP from Communications Link
LINK DOWN from Communications Link
Operating
entry / Send OPEN to Pumps
exit / Send CLOSE to Pumps
THRESHOLD EXCEEDED from Detector
Emergency
entry / Send ALARM MESSAGE to Remote Central Facility
exit / Reset Alarm
RESTART from Remote Central Facility
RESTART from Remote Central Facility
SHUTDOWN from Remote Central Facility
Disabled
THRESHOLD EXCEEDED from Detector

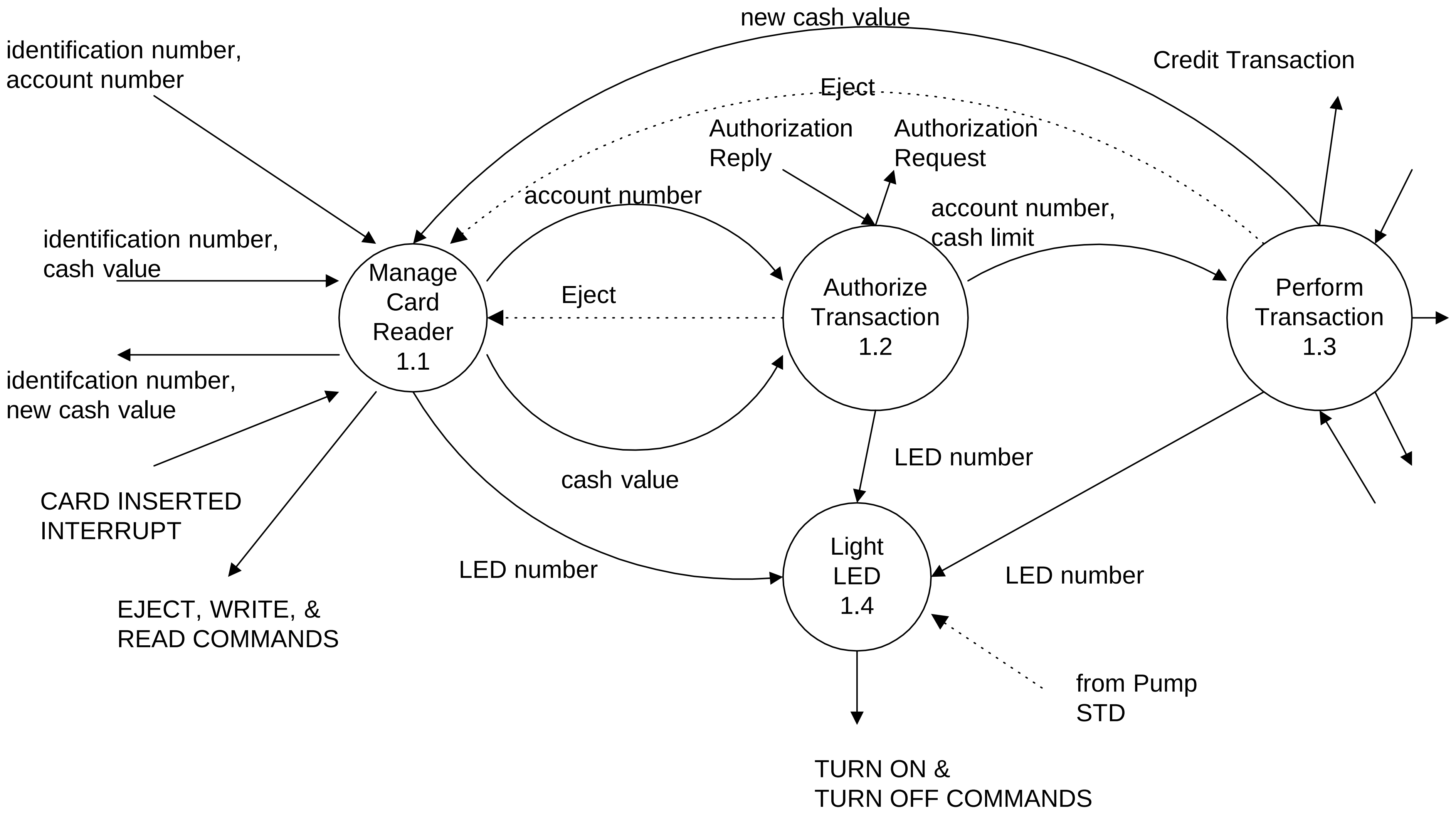
D-13 GAS STATION STATE TRANSITION DIAGRAM


Below Threshold
do: monitor sensor
exit / Sound Alarm, Send THRESHOLD EXCEEDED to Gas Station
THRESHOLD CROSSED COMING DOWN
THRESHOLD CROSSED GOING UP
Above Threshold
do: monitor sensor

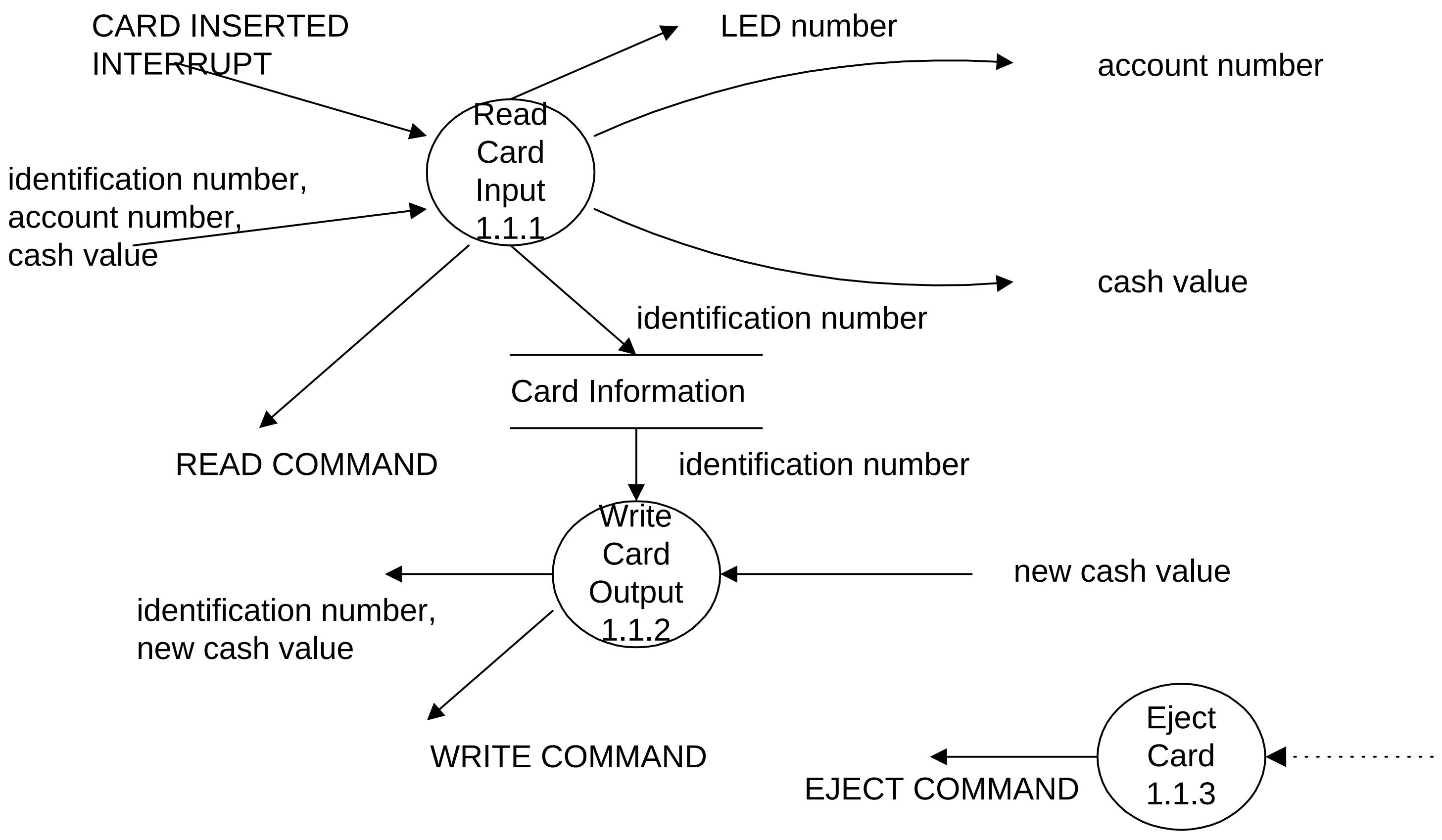
D-14 DETECTOR STATE TRANSITION DIAGRAM

AUTOMATED GAS STATION MANAGER

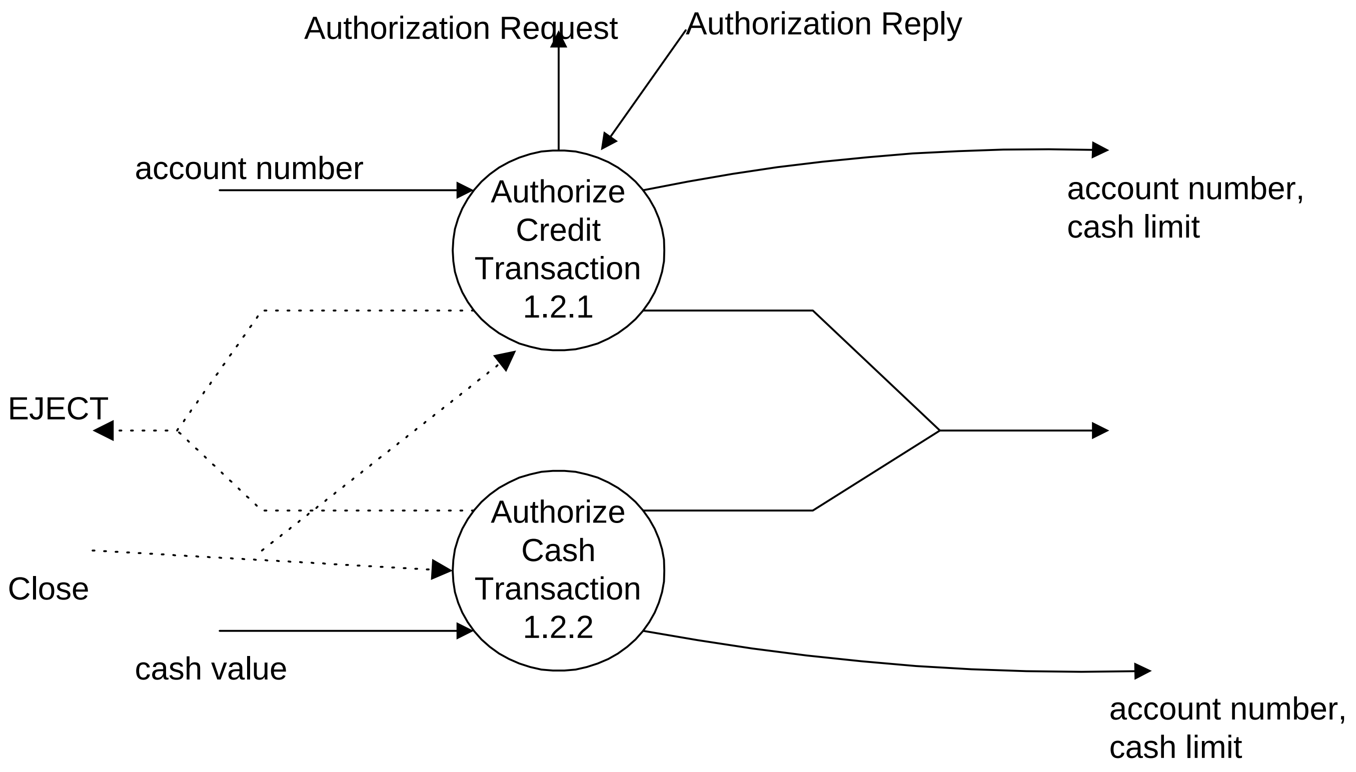
FUNCTIONAL MODEL



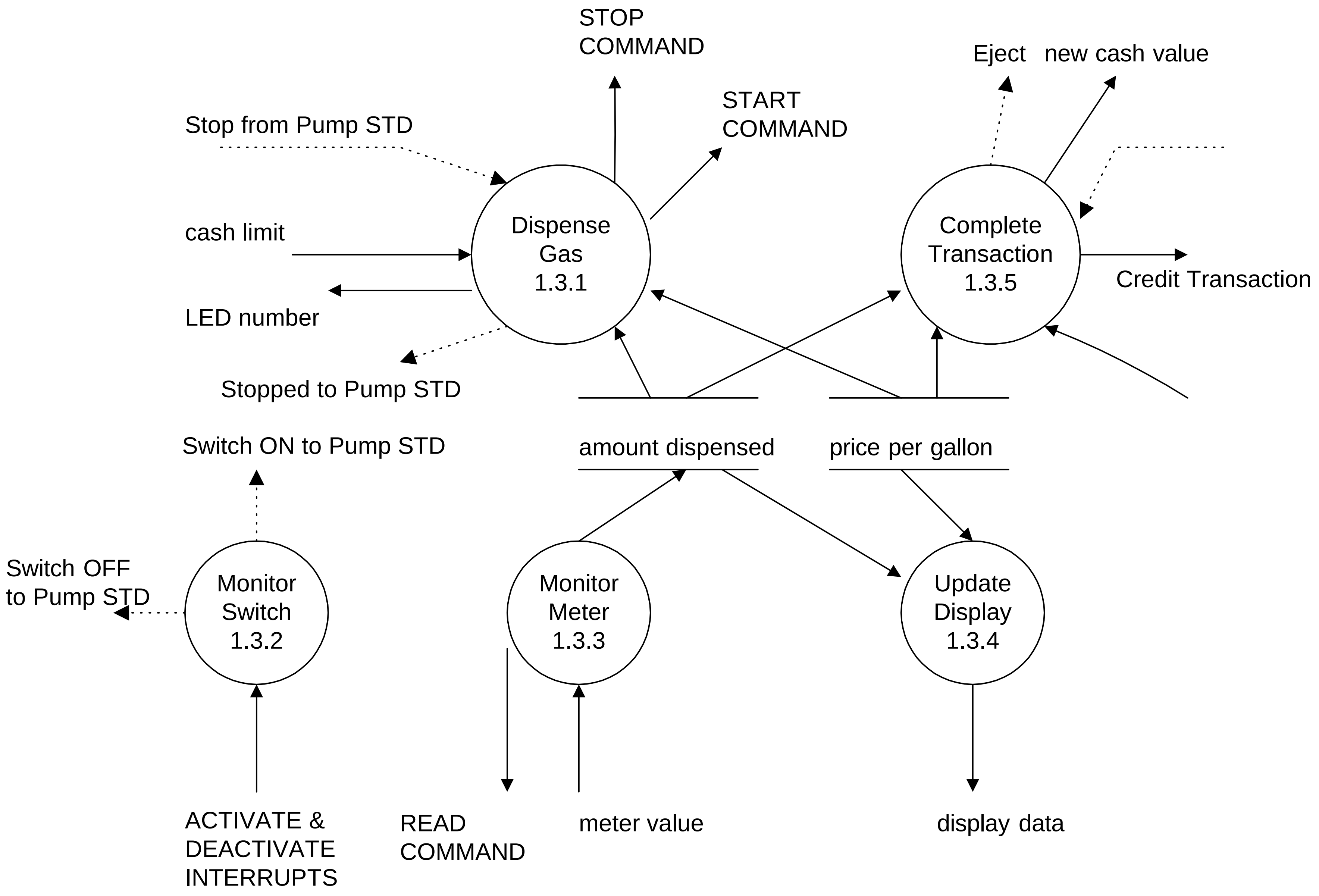
D-19 1 Process Customer DATA FLOW DIAGRAM



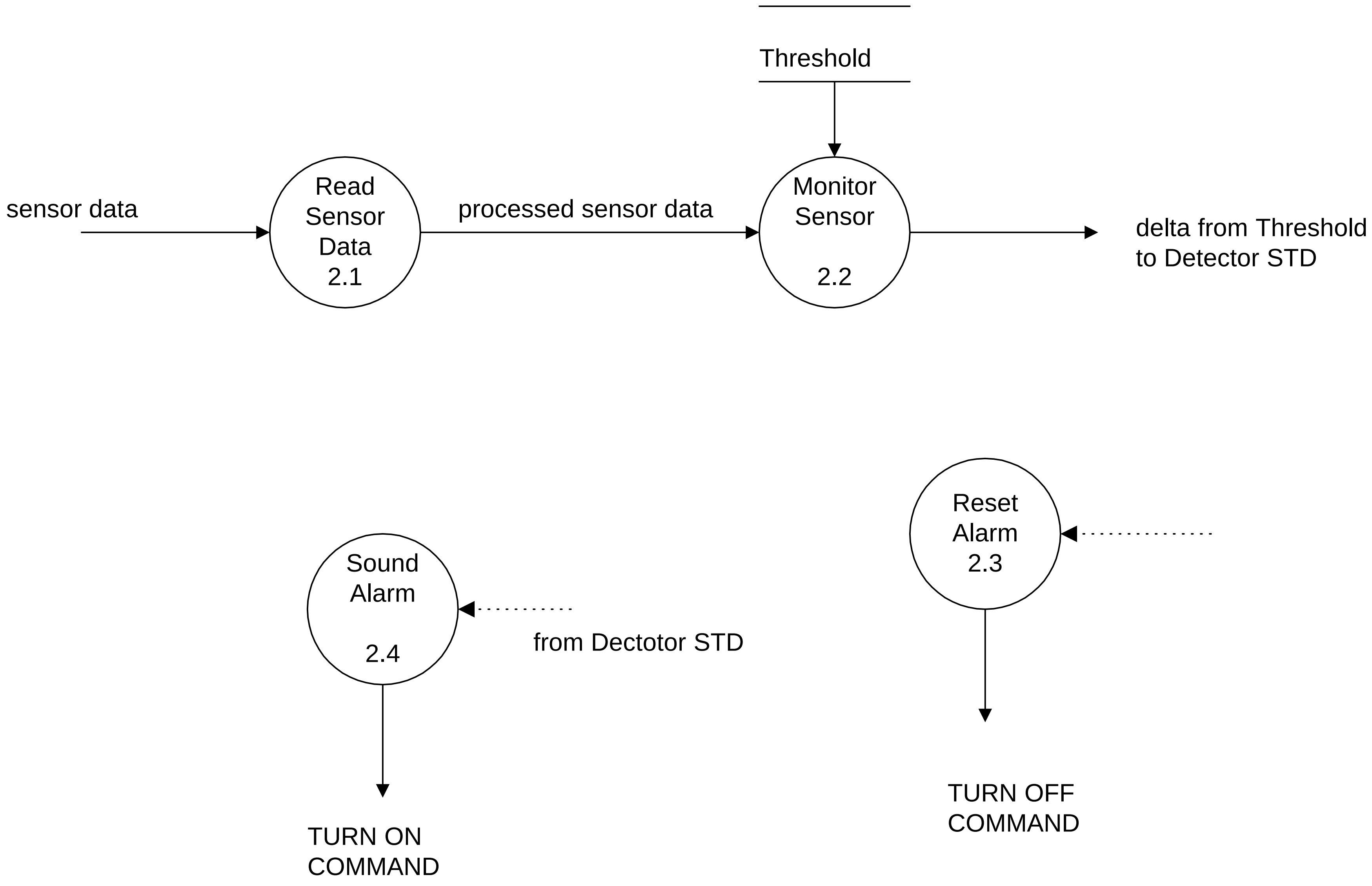
D-20 1.1 Manage Card Reader DATA FLOW DIAGRAM


Authorization Request
Authorization Reply
account number
Authorize Credit Transaction 1.2.1
account number, cash limit
EJECT
Authorize Cash Transaction 1.2.2
Close
cash value
account number, cash limit

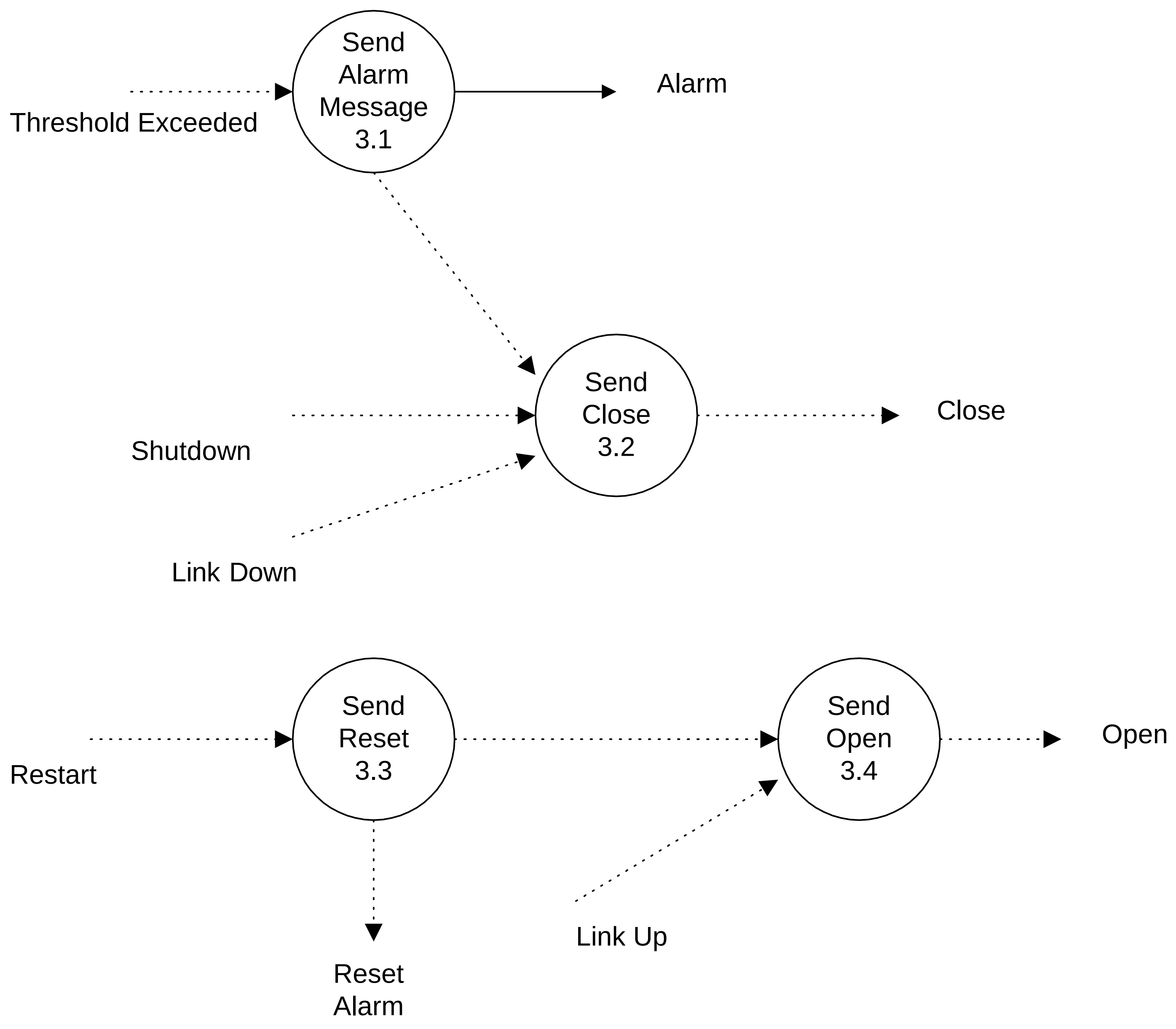
D-21 1.2 Authorize Transaction DATA FLOW DIAGRAM



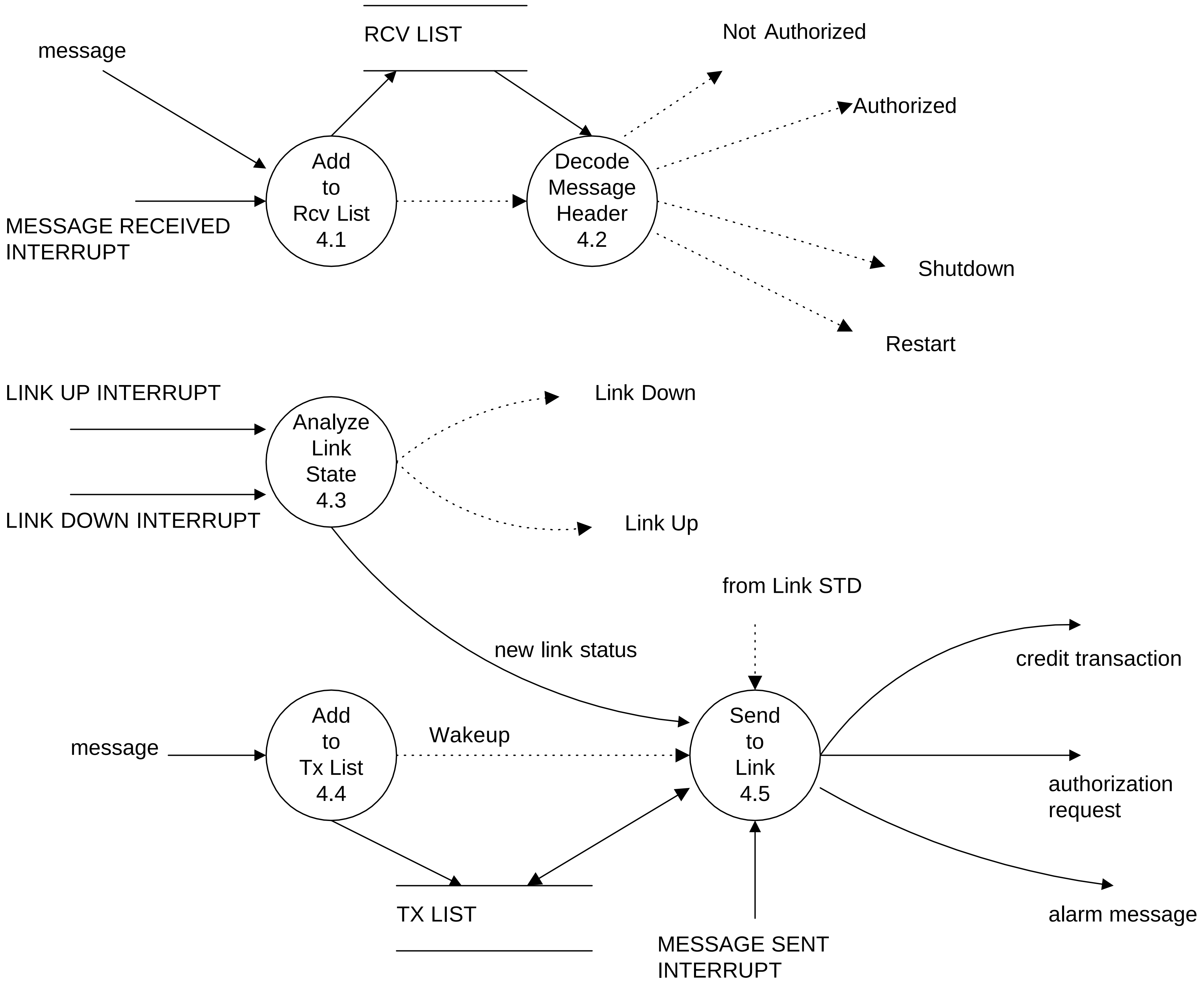
D-22 1.3 Perform Transaction DATA FLOW DIAGRAM



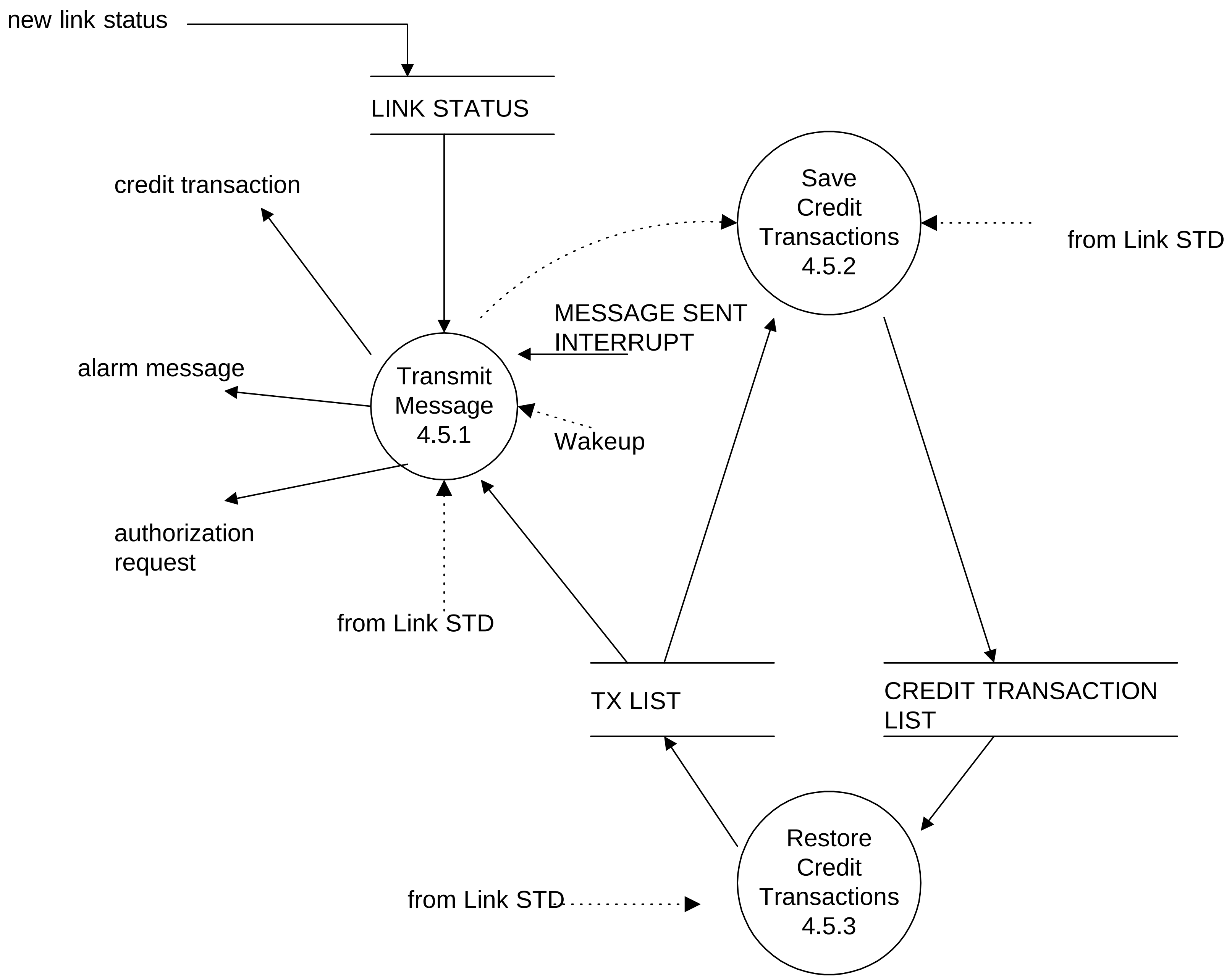
D-23 2 MANAGE ALARM DATA FLOW DIAGRAM



D-24 3 MANAGE GAS STATION DATA FLOW DIAGRAM



D-25 4 MANAGE COMMUNICATIONS LINK DATA FLOW DIAGRAM



D-26 4.5 Send to Link DATA FLOW DIAGRAM

AGSM Function Descriptions

1.1.1 Read Card Input

```
LOOP FOREVER
    WAIT for CARD INSERTED INTERRUPT
    READ identification number and store in Card Information
    IF identification number is Cash Card
        THEN READ cash value,
             store cash value in Card Information,
             CALL Authorize Cash Transaction(cash value)
    ELSEIF identification number is Corporate Credit Card
        THEN READ account number,
             store account number in Card Information,
             CALL Authorize Credit Transaction(account number)
    ELSE CALL Light  LED(Cannot Process Card)
        CALL Eject Card
    ENDIF
END LOOP
```

1.1.2 Write Card Output(new cash value)

```
WRITE(identification number from Card Information, new cash value)
    to CARD
```

1.1.3 Eject Card

```
EJECT
```

1.2.1 Authorize Credit Transaction(account number)

```
Authorization Request := BUILD (Remote Central Facility Address, Station ID, Pump ID,
                                    account number)
CALL Add to Tx List(Authorization Request)
WAIT for Authorization Reply OR Close
IF Authorization Reply  with Authorization
    THEN CALL Dispense Gas(account number,(cash limit := none))
ELSEIF Authorization Reply without Authorization
    THEN CALL Light LED(Cannot Process Card)
        CALL Eject Card
ELSE  CALL Eject Card
ENDIF
```

1.2.2 Authorize Cash Transaction(cash value)

```
IF cash value is not positive
    THEN CALL Light LED(Cash Value Used)
         CALL Eject Card
ELSE CALL Dispense Gas((account number := not valid), (cash limit := cash value))
ENDIF
```

1.3.1 Dispense Gas(cash limit)

```
Send START COMMAND to Gas Dispenser
LOOP FOREVER
    Check for Stop from Pump STD
    If Stop received THEN BREAK LOOP
    CALL Monitor Meter
    CALL Update Display
    If amount dispensed/100 * price per gallon
        >= cash limit THEN BREAK LOOP
END FOREVER LOOP
Send STOP COMMAND to Gas Dispenser
CALL Monitor Meter
CALL Update Display
RETURN
```

1.3.2 Monitor Switch

```
LOOP FOREVER
WAIT for ACTIVATE INTERRUPT OR DEACTIVATE INTERRUPT
IF ACTIVATE INTERRUPT
    THEN Send Switch ON to Pump STD
    ELSE  Send Switch OFF to Pump STD
ENDIF
END  FOREVER LOOP
```

1.3.3 Monitor Meter

```
READ meter value
Convert to hundreths of gallons
Set amount dispensed to converted value
```

1.3.4 Update Display

```
display gallons := amount dispensed/100
display cost := display gallons * price per gallon
OUTPUT display gallons to GALLONS DISPLAY
OUTPUT display cost to COST DISPLAY
```

1.3.5 Complete Transaction(account number, cash limit)

```
IF account number is valid
        THEN Credit Transaction := BUILD(Remote Central Facility Address, Station ID, Pump
                                         ID, account number,
                                         amount dispensed/100*price per gallon)
             CALL Add to TX List (Credit Transaction)
        ELSE new cash value := cash limit - (amount dispensed/100*price per gallon)
             CALL Write Card Output(new cash value)
ENDIF
CALL Eject Card
```

1.4 Light LED(LED number)

```
IF LED number is Cash Value Used
        THEN TURN ON Cash Value Used LED
        ELSE TURN ON Cannot Process Card LED
ENDIF
WAIT 15 Seconds
TURN OFF BOTH LEDs
```

2.1 Read Sensor Data

```
READ sensor data
convert the sensor data to processed sensor data (i.e., a REAL value)
CALL Monitor Sensor(processed sensor data)
```

2.2 Monitor Sensor(processed sensor data)

```
compute delta from Threshold using processed sensor data and Threshold
pass delta from Threshold to Detector STD
```

2.3 Reset Alarm

```
Send TURN OFF COMMAND to Alarm
```

2.4 Sound Alarm

```
Send TURN ON COMMAND to Alarm
```

3.1 Send Alarm Message

```
Alarm Message := BUILD(Remote Central Facility Address, Station ID)
CALL Add to Tx List(Alarm Message)
```

3.2 Send Close

```
LOOP FOR EVERY Pump
```

```
    Send Close to Pump
END LOOP
```

3.3 Send Reset

```
CALL Reset Alarm
```

3.4 Send Open

```
LOOP FOR EVERY PUMP
    Send Open to Pump
END LOOP
```

4.1 Add to Rcv List

```
Accept message from Link
Add message to RCV LIST
CALL Decode Message Header
```

4.2 Decode Message Header

```
IF message is a RESTART COMMAND
    THEN generate a Restart event for the Gas Station STD
ELSEIF message is a SHUTDOWN COMMAND
    THEN generate a Shutdown event for the Gas Station STD
ELSEIF message is an authorization reply
    THEN IF message contains authorization
            THEN generate an Authorized event for the appropriate
                    Pump STD
            ELSE generate an Unauthorized event for the appropriate
                    Pump STD
        ENDIF
ENDIF
Remove message from the RCV LIST
```

4.3 Analyze Link State

This function operates following the rules in the Link STD
set new link status as indicated in Link STD
generate appropriate events as indicated in Link STD

4.4 Add to Tx List(message)

```
Add message to TX LIST
CALL Transmit Message
```

4.5.1 Transmit Message

```
CALL ENTRY Transmit Message:
    IF LINK STATUS is UP  and no message is being transmitted
        THEN start transmission of next message from TX LIST
    ELSEIF LINK STATUS is DOWN
        THEN CALL Save Credit Transactions
    ENDIF
RETURN

MESSAGE SENT INTERRUPT ENTRY:
    IF TX LIST is empty
        THEN RETURN
        ELSE CALL Transmit Message
    ENDIF
RETURN
```

4.5.2 Save Credit Transactions

```
FOR EVERY message on TX LIST
    IF message is a Credit Transaction
        THEN Save message to CREDIT TRANSACTION LIST on  Disk
    ENDIF
    Remove message from TX LIST
END FOR EVERY message on TX LIST LOOP
```

4.5.3 Restore Credit Transactions

```
FOR EVERY message on CREDIT TRANSACTION LIST on Disk
    Add message to head of TX LIST
    Remove message from CREDIT TRANSACTION LIST on Disk
END FOR EVERY message on CREDIT TRANSACTION List on Disk LOOP
```

AGMS DATA DICTIONARY FOR FUNCTIONAL MODEL

A. account number: NUMERIC

-- Number issued on Corporate Credit Cards to identify the account

B. amount dispensed: REAL

--Contains the amount of gasoline dispensed at a Pump in units of hundreths of a gallon

C. Card Information: RECORD

--Holds the data read from a Card inserted into the Card Reader

1. identification number: NUMERIC
2. account number: NUMERIC
3. cash value: INTEGER

D. cash limit: INTEGER

--Contains the limit in cents that a Customer can use to buy gas
--If the value is negative, then the Customer has no limit

E. cash value: INTEGER

--Indicates the value of a Cash Card in cents.

F. CREDIT TRANSACTION LIST: FILE

--A File containing saved Credit Transaction Messages

G. delta from Threshold: REAL

--Contains a measure of the distance of a given sensor reading from the
--Threshold. (If positve, then the distance is above the Threshold. If negative,
-- then the distance is below the Threshold.)

H. display data: RECORD

--Contains two values to be displayed

1. display gallons: FIXED POINT (Gallons and Tenths and Hundreths)
2. display cost: FIXED POINT (Dollars and Cents)

I. identification number: NUMERIC

--Distinguishes a Cash Card from a Corporate Credit Card

J. LED number: INTEGER

--Identifies the LED on a Pump that is to be turned on.

--	Values:	0 is Cash Value Used
--		1 is Cannot Process Card

K. LINK STATUS: INTEGER

--Indicates the current state of the Communications Link

--	Values:	0x00 is Link Up
--		0x01 is Link Down

L. messages: RECORD

--The Data Structure that holds a message

1. To: ADDRESS (Station or Remote Central Facility)
2. From: ADDRESS (Station or Remote Central Facility)
3. Type: BYTE
    - 0 is SHUTDOWN COMMAND
    - 1 is RESTART COMMAND
    - 2 is credit transaction
    - 3 is alarm message
    - 4 is authorization request
    - 5 is authorization reply
4. Optional additional parameters determined by Type.
    - For credit transaction:
        - Pump ID: INTEGER
        - account number: NUMERIC
        - cost of gas: INTEGER (in cents)
    - For authorization request
        - Pump ID: INTEGER
        - account number: NUMERIC
    - For authorization reply
        - Pump ID: INTEGER
        - response: BOOLEAN
            - 0 is Not Authorized
            - 1 is Authorized

M. meter value: REAL

-- Data read from the Gas Dispenser's meter

N. new cash value: INTEGER

-- The value (in cents) to place back onto the Cash Card after the transaction is complete

O. new link status: INTEGER

-- The new status of the communications link. Values can be 0, link up, or 1, link down

P. price per gallon: INTEGER

-- The cost of gasoline, per gallon, at the Pump. The units are cents. This value is a
-- configuration setting for each Pump.

Q. processed sensor data: REAL

-- The current value read from a sensor, but converted to a real number.
-- For a smoke detector, this measures smoke particle concentration in parts per million.
-- For a heat detector, this measrues temperature in degrees Celsius.

R. RCV LIST: LINKED_LIST

-- This is the queue of messages received on the Communications Link, but not yet
-- processed.

S. sensor data: DEVICE DEPENDENT

-- The raw input from a sensor device. The form is dependent on the specific sensor.

T. Threshold: REAL

-- The value that a processed sensor data input is compared against.
-- This is set as a configuration option on each sensor.

U. TX LIST: LINKED_LIST

-- The queue of messages that are to be transmitted, but that have not yet been transmitted

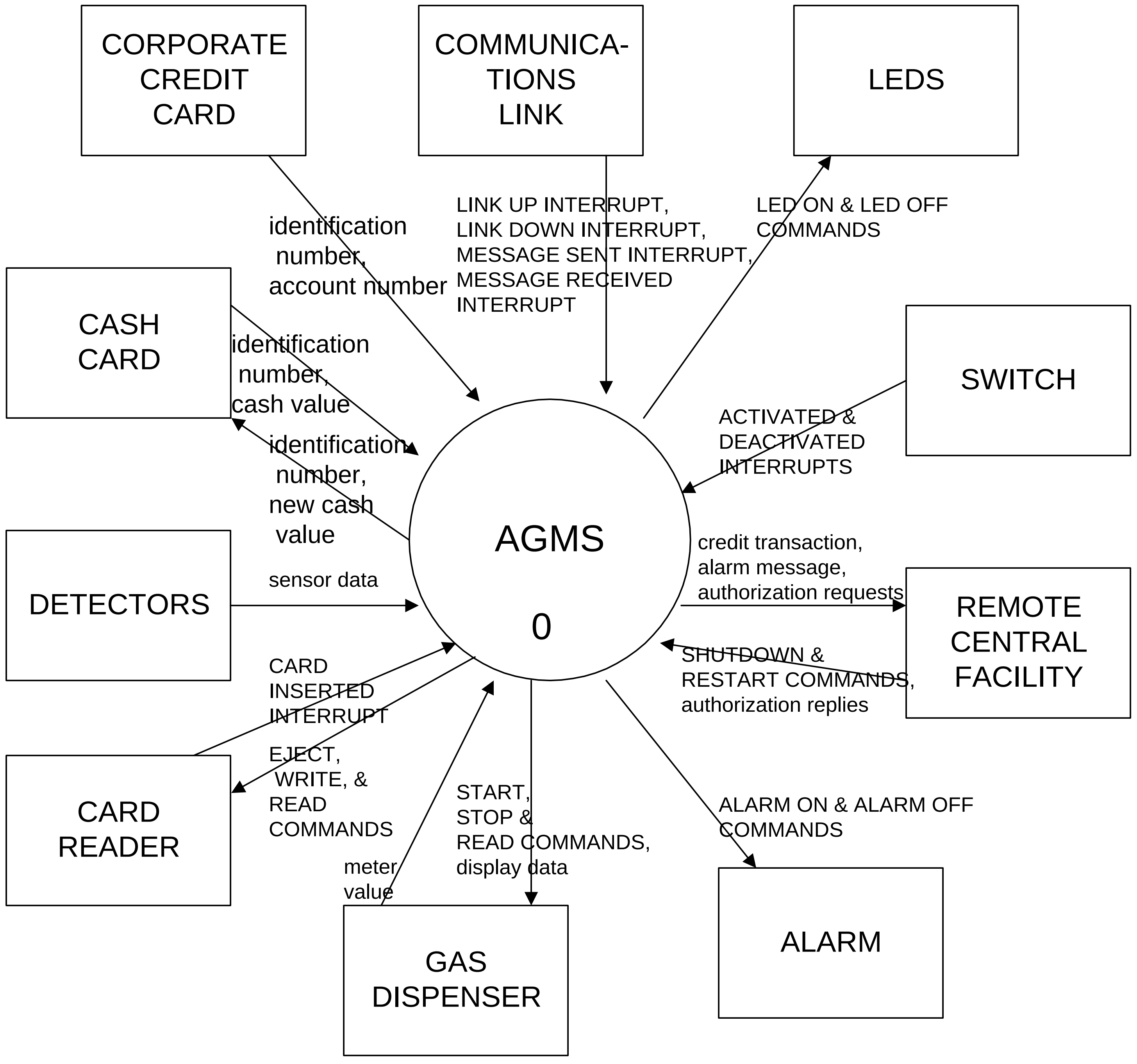
AUTOMATED GAS STATION MANAGER

OBJECT COMMUNICATION DIAGRAMS,

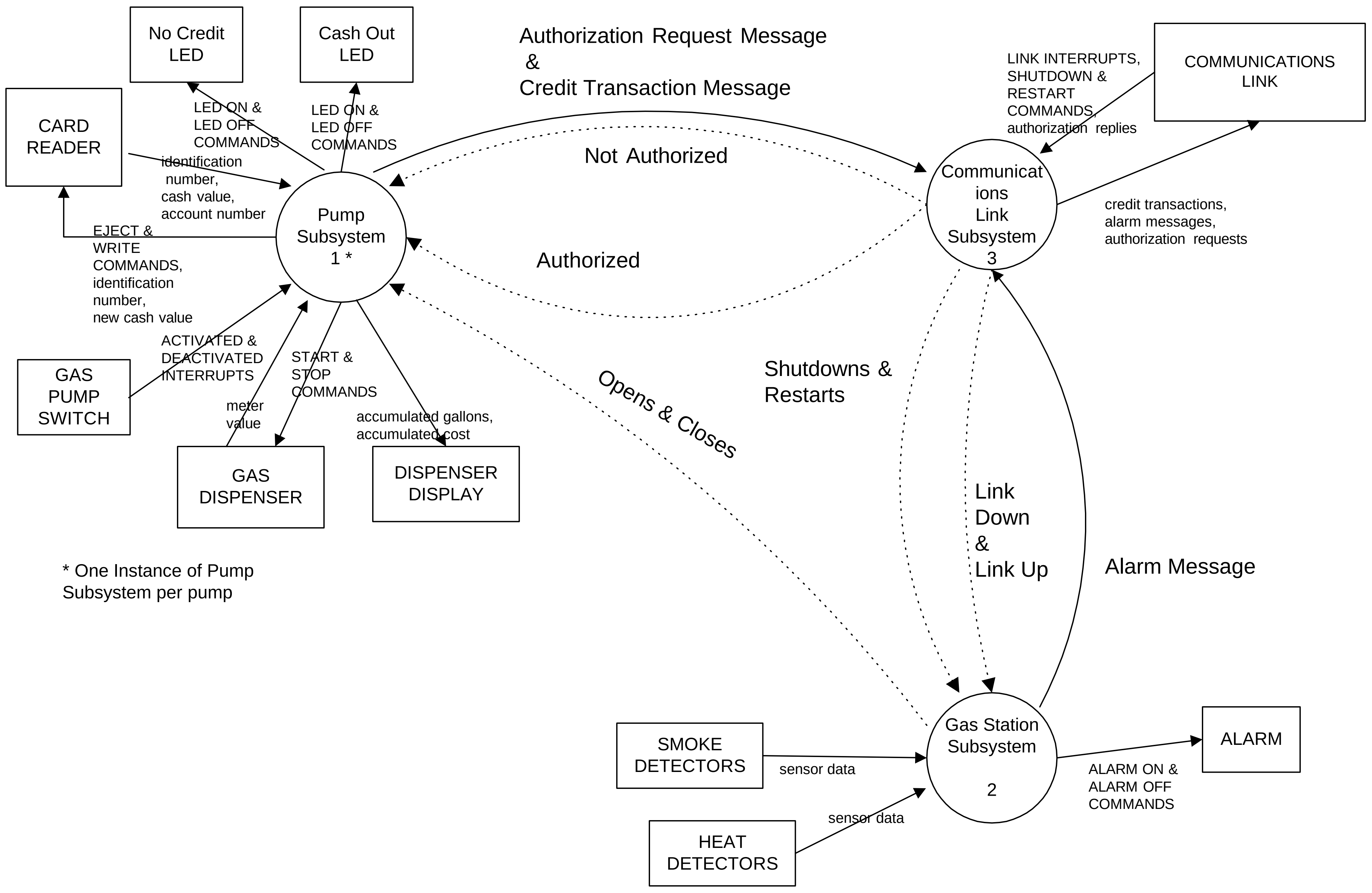
OBJECT FUNCTION DIAGRAMS,

AND

OBJECT FUNCTION DESCRIPTIONS

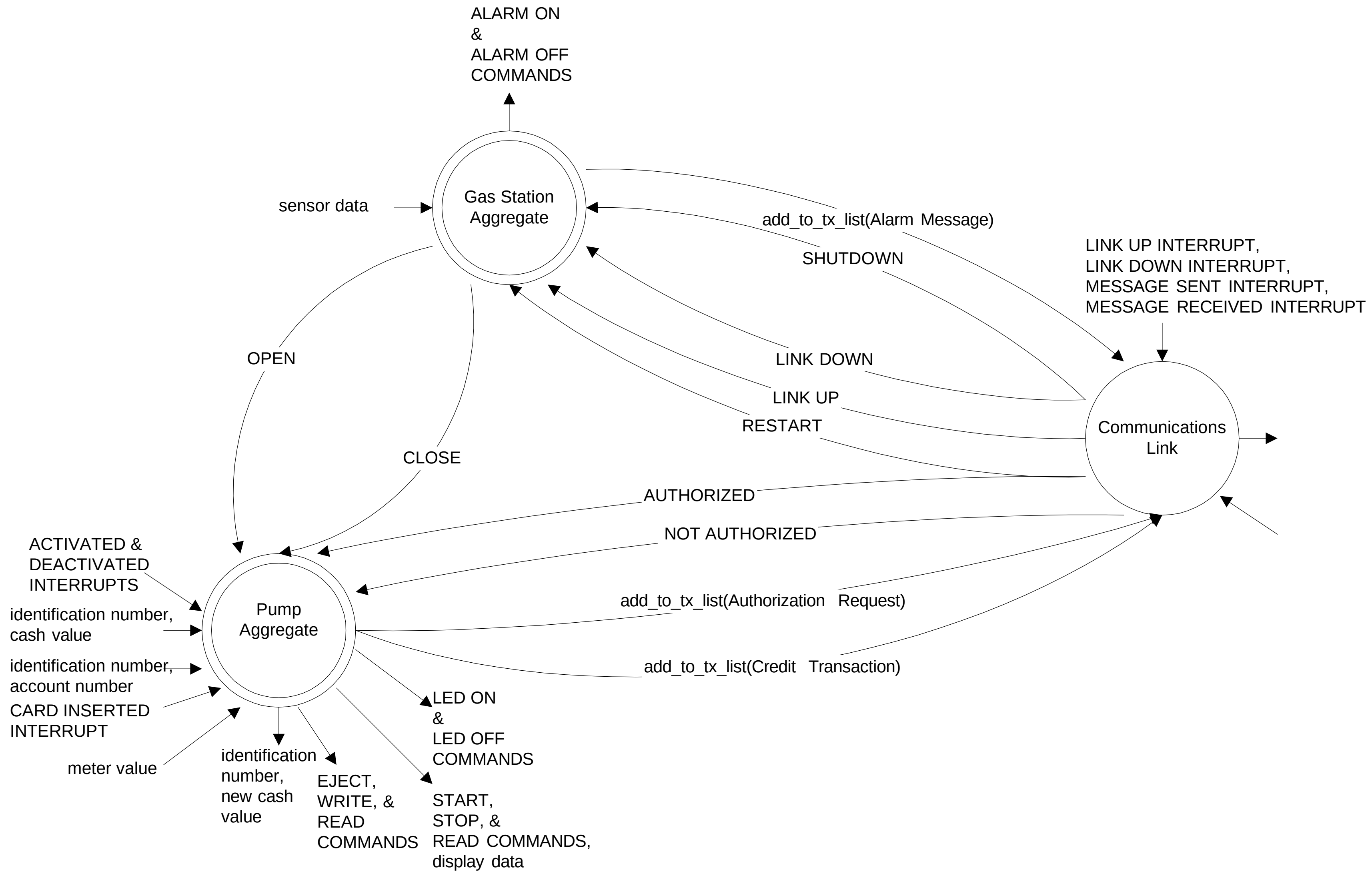

CORPORATE CREDIT CARD
COMMUNICA-TIONS LINK
LEDS
CASH CARD
SWITCH
DETECTORS
REMOTE CENTRAL FACILITY
CARD READER
GAS DISPENSER
ALARM
AGMS 0
identification number, account number
LINK UP INTERRUPT, LINK DOWN INTERRUPT, MESSAGE SENT INTERRUPT, MESSAGE RECEIVED INTERRUPT
LED ON & LED OFF COMMANDS
identification number, cash value
identification number, new cash value
ACTIVATED & DEACTIVATED INTERRUPTS
sensor data
credit transaction, alarm message, authorization requests
SHUTDOWN & RESTART COMMANDS, authorization replies
CARD INSERTED INTERRUPT
EJECT, WRITE, & READ COMMANDS
START, STOP & READ COMMANDS, display data
meter value
ALARM ON & ALARM OFF COMMANDS

D-36 AGMS CONTEXT DIAGRAM



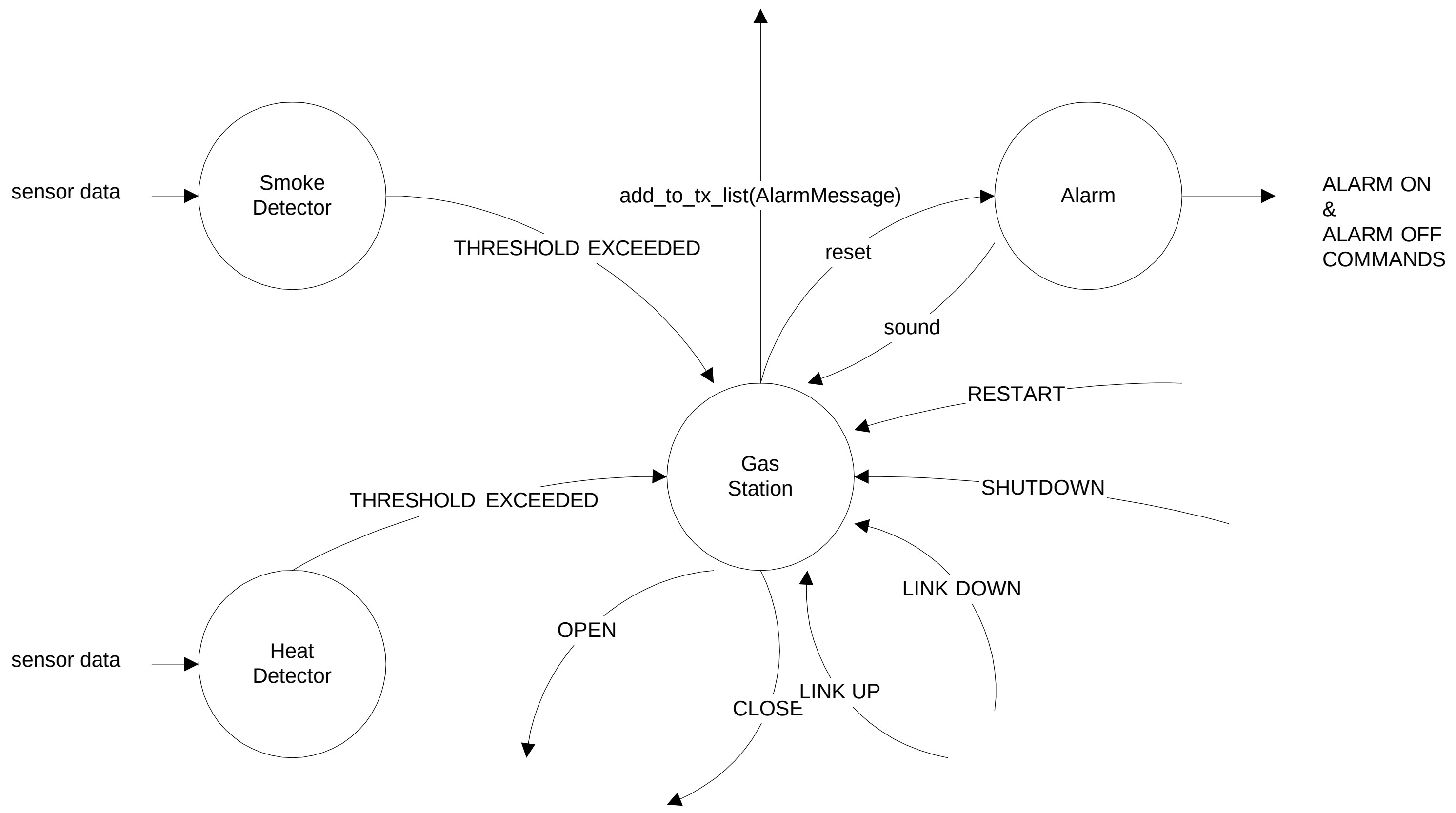
Authorization Request Message & Credit Transaction Message
No Credit LED
Cash Out LED
CARD READER
COMMUNICATIONS LINK
LINK INTERRUPTS, SHUTDOWN & RESTART COMMANDS, authorization replies
LED ON & LED OFF COMMANDS
LED ON & LED OFF COMMANDS
identification number, cash value, account number
Not Authorized
Communications Link Subsystem 3
Pump Subsystem 1 *
credit transactions, alarm messages, authorization requests
EJECT & WRITE COMMANDS, identification number, new cash value
Authorized
ACTIVATED & DEACTIVATED INTERRUPTS
START & STOP COMMANDS
GAS PUMP SWITCH
meter value
accumulated gallons, accumulated cost
Opens & Closes
Shutdowns & Restarts
GAS DISPENSER
DISPENSER DISPLAY
Link Down & Link Up
Alarm Message
* One Instance of Pump Subsystem per pump
Gas Station Subsystem 2
SMOKE DETECTORS
sensor data
ALARM
ALARM ON & ALARM OFF COMMANDS
sensor data
HEAT DETECTORS

AGMS SUBSYSTEM DECOMPOSITON

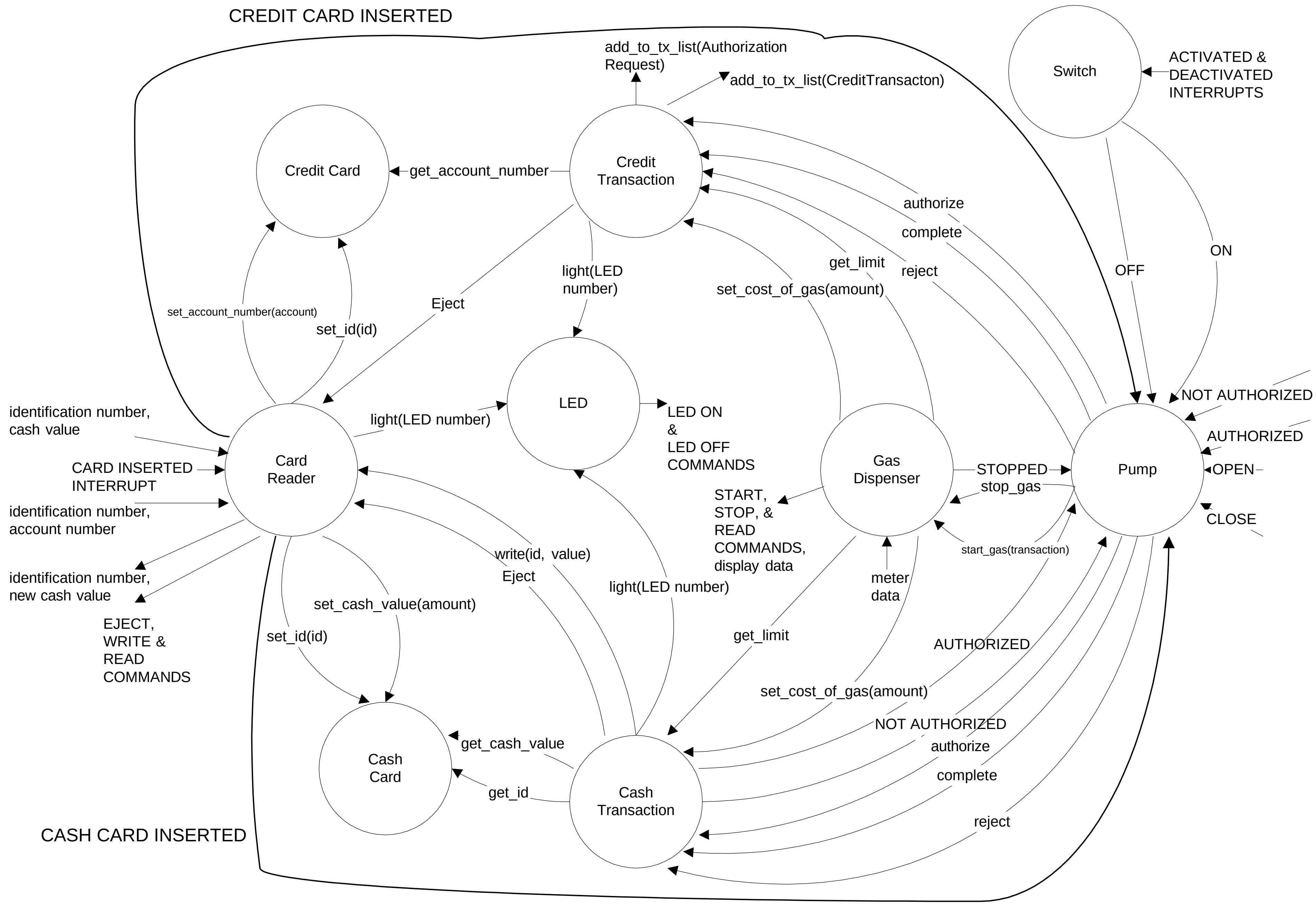


ALARM ON & ALARM OFF COMMANDS
sensor data
Gas Station Aggregate
add_to_tx_list(Alarm Message)
SHUTDOWN
LINK UP INTERRUPT, LINK DOWN INTERRUPT, MESSAGE SENT INTERRUPT, MESSAGE RECEIVED INTERRUPT
OPEN
LINK DOWN
LINK UP
RESTART
CLOSE
Communications Link
AUTHORIZED
NOT AUTHORIZED
ACTIVATED & DEACTIVATED INTERRUPTS
identification number, cash value
Pump Aggregate
add_to_tx_list(Authorization Request)
identification number, account number
add_to_tx_list(Credit Transaction)
CARD INSERTED INTERRUPT
LED ON & LED OFF COMMANDS
meter value
identification number, new cash value
EJECT, WRITE, & READ COMMANDS
START, STOP, & READ COMMANDS, display data

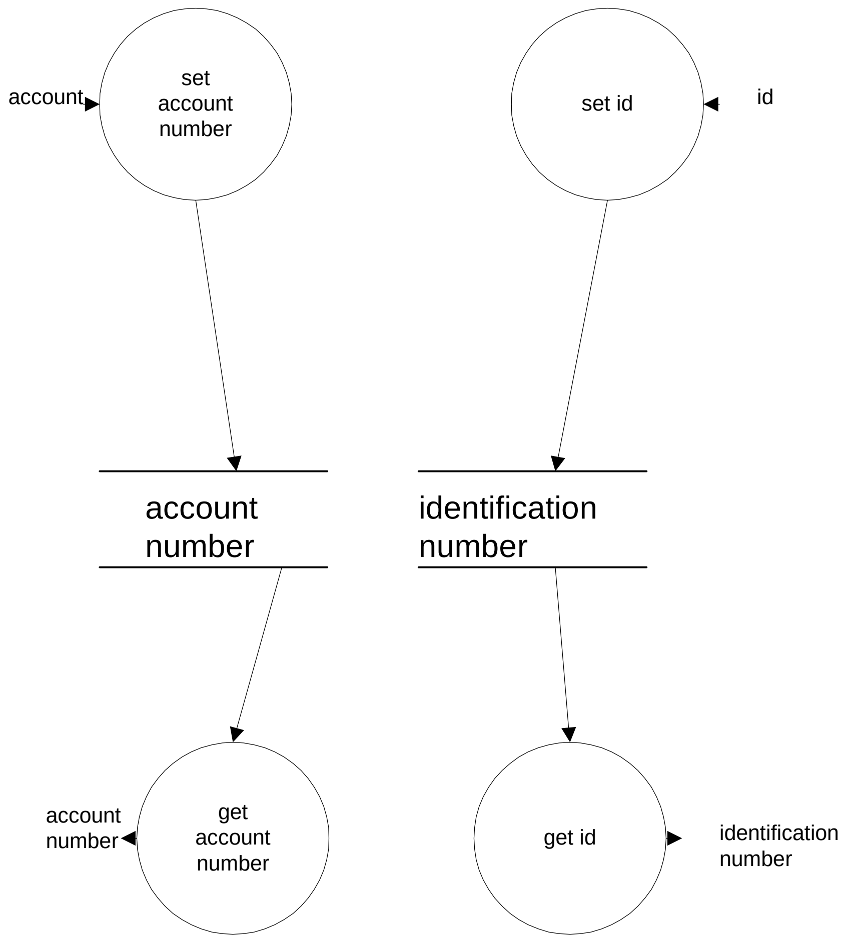
D-37 TOP-LEVEL OBJECT COMMUNICATIONS DIAGRAM



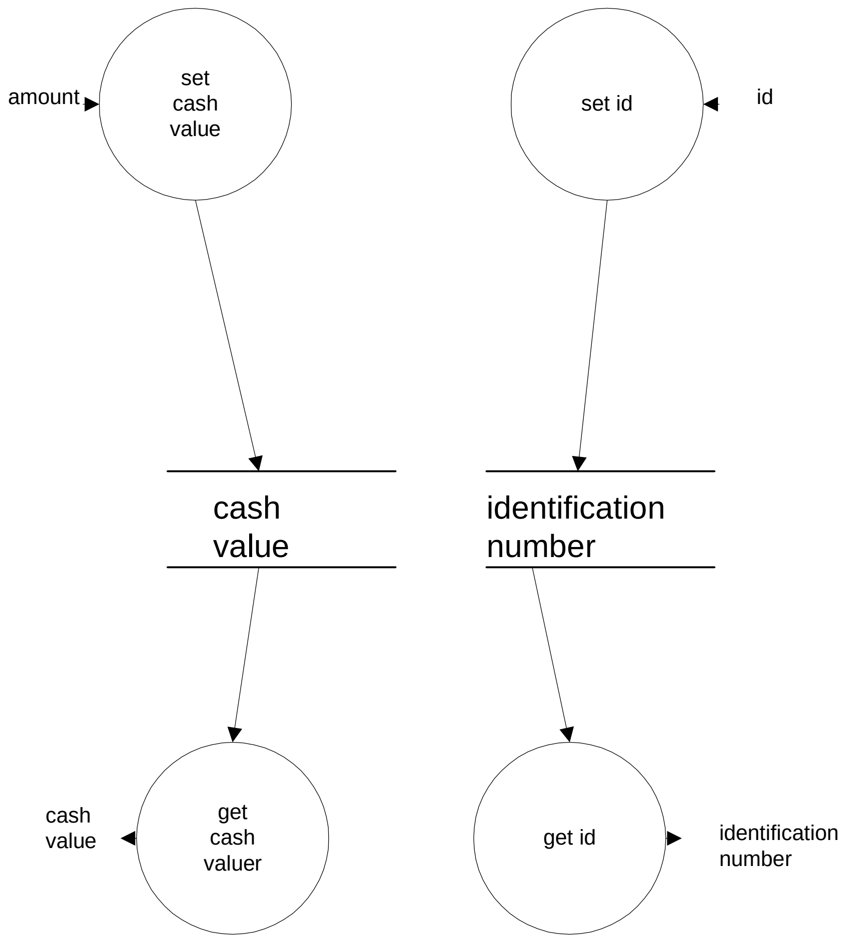
D-38 GAS STATION AGGREGATE OBJECT COMMUNICATIONS DIAGRAM



D-39 PUMP AGGREGATE OBJECT COMMUNICATION DIAGRAM



D-42 CREDIT CARD OBJECT FUNCTION DIAGRAM



D-42 CASH CARD OBJECT FUNCTION DIAGRAM

AGSM Object Function Descriptions

Card Reader Object

```
write(id, new cash value)
        WRITE id and new cash value to Cash Card
end_write

Eject
        SEND EJECT COMMAND to CARD READER
end_eject

read
        READ identification number
        IF identification number is for Cash Card
                THEN Cash Card.create,
                        Cash Card.set_id(identification number)
                        READ cash value,
                        Cash Card.set_cash_value(cash value),
                        SEND CASH CARD INSERTED EVENT to Pump
        ELSEIF identification number is for Corporate Credit Card
                THEN Corporate Credit Card.create,
                        Corporate Credit Card.set_id(identification number)
                        READ account number,
                        Corporate Credit Card.set_account_number(account number),
                        SEND CREDIT CARD INSERTED EVENT to Pump
                ELSE    LED.light(Cannot Process Card)
                        Eject
        ENDIF
end_read
```

Card Object

```
set_id_number(id)
        identification number := id
end_set_id_number

get_id_number()
        RETURN identification number
end_get_id_number

set_account_number(x)
        RETURN
end_set_account_number
```

```
get_account_number()
    RETURN end_get_account_number
end_get_account_number

set_cash_value(x)
    RETURN invalid account number
end_set_cash_value

get_cash_value()
    RETURN zero
end_get_cash_value
```

Credit Card Object

```
set_account_number(account)
    account number := account
end_set_account_number

get_account_number()
    RETURN account number
end_get_account_number
```

Cash Card Object

```
set_cash_value(value)
    cash value := value
end_set_cash_value

get_cash_value()
    RETURN cash value
end_get_cash_value
```

Transaction Object

```
get_limt()
    RETURN limit
end_get_limt

set_cost_of_gas(amount:INTEGER)
    IF amount < 0
        THEN cost of gas := 0
    ELSE
        cost of gas := amount
    ENDIF
end_set_cost_of_gas
```

Credit Transaction Object

```
authorize(Credit Card)
    limt : = 0
    Authorization Request := BUILD (Remote Central Facility Address, Station ID, Pump
ID,                                             Credit Card.get account number)
    Communications Link.Add_to_Tx_List(Authorization Request)
end_authorize

reject()
    LED.light(Cannot Process Card)
    Card Reader.Eject
end_reject

complete(Credit Card)
    Credit Transaction := BUILD(Remote Central Facility Address, Station ID, Pump
                                    ID, Credit Card.get_account_number,
                                    cost_of_gas)
    Communications Link.Add_to_TX_List (Credit Transaction)
    Card Reader.Eject
end_complete
```

Cash Transaction Object

```
authorize(Cash Card)
    IF Cash Card.get_cash_value > 0
        THEN  limit := Cash Card.get_cash_value
              GENERATE AUTHORIZED for Pump
        ELSE  GENERATE NOT AUTHORIZED for Pump
end_authorize

reject()
    LED.light(Cash Value Used)
    Card Reader.Eject
end_reject

complete(Cash Card)
    new cash value := Cash Card.get_cash_value - cost of gas
    Card Reader.write(Cash Card.get_id, new cash value)
    Card Reader.Eject
end_complete
```

Gas Dispenser Object

```
clear_amount_dispensed()
    OUTPUT BLANK COMMAND to GALLONS DISPLAY
    OUTPUT BLANK COMMAND to COST DISPLAY
```

```
        amount_dispensed := zero
end_clear_amount_dispensed

update_amount_dispensed()
        READ meter value
        Convert to hundreths of gallons
        Set amount dispensed to converted value
end_supdate_amount_dispensed

update_display()
        display gallons := amount dispensed/100
        display cost := display gallons * price per gallon
        OUTPUT display gallons to GALLONS DISPLAY
        OUTPUT display cost to COST DISPLAY
end_update_display

get_amount_dispensed()
        RETURN amount_dispensed
end_get_amount_dispensed

get_price()
        RETURN price per gallon
end_get_price

?limit_reached(Transaction)
        IF Transaction.get_limit = 0 THEN RETURN FALSE
        IF amount dispensed/100 * price per gallon >= Transaction.get_limit
                THEN RETURN TRUE
                ELSE RETURN FALSE
end_?limit_reached

start_gas()
        SEND START COMMAND to Gas Dispenser
end_start_gas

stop_gas()
        SEND STOP COMMAND to Gas Dispenser
        update_amount_dispensed
        update_display
        Transaction.set_cost_of_gas(amount_dispensed * 100 / price_per_gallon)
        SEND STOPPED  to Pump
end_stop_gas
```

LED Object

```
light(LED number)
    IF LED number is Cash Value Used
        THEN LED ON Cash Value Used LED
             WAIT 10 Seconds
             LED OFF Cash Value Used LED
        ELSE LED ON Cannot Process Card LED
             WAIT 10 Seconds
             LED OFF Cannot Process Card LED
    ENDIF
end_light
```

Detector Object

```
monitor_sensor():REAL
    compute delta from Threshold using processed sensor data and Threshold
    RETURN delta
end_monitor_sensor

send_threshold_exceeded()
    GENERATE THRESHOLD EXCEEDED for Gas Station
end_send_threshold_exceeded

process_change(delta)
    ENCAPSULATES DETECTOR STD
end_process_chage
```

Heat Detector Object

```
read()
    READ sensor data from temperature detector
    convert the sensor data to processed sensor data (i.e., a REAL value)
end_read()
```

Smoke Detector Object

```
read()
    READ sensor data from particle counter
    convert the sensor data to processed sensor data (i.e., a REAL value)
end_read()
```

Pump Object

```
process_event(EVENT)
    ENCAPSULATES THE PUMP STATE TRANSITION DIAGRAM
end_process_event
```

Switch Object

```
handle_switch_interrupt()
    EVENT := READ interrupt register
    process_event(EVENT)
end_handle_switch_interrupt

process_event(event)
    ENCAPSULATES SWITCH STATE TRANSITION DIAGRAM
end_process_event

send_switch_on
    GENERATE ON  for Pump
end_send_switch_on

send_switch_off
    GENERATE OFF for Pump
end_send_switch_off
```

Alarm Object

```
sound()
    OUTPUT ALARM ON COMMAND to Alarm
end_sound

reset()
    OUTPUT ALARM OFF COMMAND to Alarm
end_reset
```

Gas Station Object

```
process_event(EVENT)
    ENCAPSULATES the Gas Station STATE TRANSITION DIAGRAM
end_process_event

send_alarm_message()
    Alarm Message := BUILD(Remote Central Facility Address, Station ID)
    Communications Link.Add_to_Tx_List(Alarm Message)
end_send_alarm_message

send_close()
    LOOP FOR EVERY Pump
        GENERATE CLOSE for Pump
    END LOOP
end_send_close

send_open()
    LOOP FOR EVERY Pump
```

```
        GENERATE OPEN for Pump
    END LOOP
end_send_open
```

Communications Link Object

```
handle_link_state_interrupt()
    EVENT := INPUT INTERRUPT VALUE
    analyze_link(EVENT)
end_link_state_interrupt

handle_message_received_interrupt()
    POINTER := INPUT MESSAGE BUFFER ADDRESS
    add_to_rcv_list(POINTER)
end_handle_message_received_interrupt

handle_message_send_interrupt()
    POINTER := MESSAGE BUFFER ADDRESS
    FREE(POINTER)
    transmit_message
end_handle_message_send_interrupt

add_to_rcv_list(message)
    Insert message at tail of RCV_LIST
end_add_to_rcv_list

decode_message()
    get message from head of RCV_LIST
    IF message is a RESTART COMMAND
        THEN GENERATE a RESTART for the Gas Station
    ELSEIF message is a SHUTDOWN COMMAND
        THEN GENERATE a SHUTDOWN for the Gas Station
    ELSEIF message is an authorization reply
        THEN IF message contains authorization
                THEN GENERATE an AUTHORIZED for the appropriate
                        Pump
                ELSE GENERATE a NOT AUTHORIZED for the appropriate
                        Pump
            ENDIF
    ENDIF
    Remove message from the RCV LIST
    FREE message for a new receive buffer
end_decode_message


analyze_link_state(EVENT)
```

```
    ENCAPSULATES Link STATE TRANSITION DIAGRAM
end_analyze_link_state

add_to_tx_list(message)
    Insert Message at tail of TX LIST
    transmit_message
end_add_to_tx_list

transmit_message()
    IF TX_LIST is empty THEN RETURN
    IF status is UP  and no message is being transmitted
        THEN start transmission of next message from TX LIST
    ELSEIF LINK STATUS is DOWN
        THEN save_credit_transactions
    ENDIF
    RETURN
end_transmit_message

save_credit_transactions()
    FOR EVERY message on TX LIST
        IF message is a Credit Transaction
            THEN Save message to CREDIT TRANSACTION LIST on  Disk
        ENDIF
        Remove message from TX LIST
    END FOR EVERY message on TX LIST LOOP
end_save_credit_transactions

restore_credit_transactions()
    FOR EVERY message on CREDIT TRANSACTION LIST on Disk
        Add message to head of TX LIST
        Remove message from CREDIT TRANSACTION LIST on Disk
    END FOR EVERY message on CREDIT TRANSACTION List on Disk LOOP
end_restore_credit_transaction
```

Overview Of AGSM Task Architecture

The Automated Gas Station Management (AGSM) software comprises a set four tasks to manage the gas station and a set of four tasks for each pump. This architecture is illustrated in Figure E-3. The architecture is reproduced in Figure E-4 with the inter-task communications identified by numbering the arcs between the tasks.

As shown in Figure E-3, each gas station consists of four tasks: 1) Gas Station Control, 2) Detector Array, 3) Alarm, and 4) Communications Link. The Gas Station Control task, activated by the arrival of messages in the Gas Station Control Queue, manages the operation of the gas station software. The Detector Array task, activated by a periodic timer event, polls each of the smoke and heat detectors installed at the gas station. When the concentration of smoke particles or the temperature exceeds a pre-defined threshold at any of the smoke or heat detectors, respectively, the Detector Array task notifies the Gas Station Control task. The Alarm task, when enabled, continuously sounds the gas station alarm until the task is disabled. The Communications Link task acts as a conduit between the gas station and a remote central facility. Messages sent from the station to the remote central facility and messages received at the station from the remote cental facility pass through the Communications Link task. The task is activated by messages arriving in the Transmit Messages Queue and by any of three hardware interrupts (message sent, message received, and link state change).

The gas station includes a number of pumps, each of which is controlled through four software tasks, as shown in Figure E-3. The main task managing a pump is the Pump Control task which sequences and synchronizes external inputs that affect the pump. The Pump Control task is activated by the arrival of a message in the Pump Control Queue. The Gas Dispenser task, activated by a message from the Pump Control task, controls the dispensing of gasoline through the pump's gas dispenser. The Card Reader task, when activated by a Card Inserted Interrupt, extracts information from a customer's cash or credit card, updates the Card data abstraction IHM, and awaits further commands issued on behalf of the Pump Control task. The Switch task simply converts switch activation and deactivation interrupts into internal events and sends them to the Pump Control Queue.

Turning attention to Figure E-4, some typical flows of control through the system can be described. A customer approaches an operating pump at an open gas station and inserts a credit or cash card causing a Card Inserted Interrupt (1). The Card Reader task determines the type of card that was inserted. If the card is not recognized, then the card is ejected from the card reader. If the card is recognized, an appropriate data abstraction object is created and the information from the card is stored within the object (2). A card inserted event is then passed to the Pump Control Queue (3). The Pump Control task examines the card inserted event and related card information (4). If the card is a cash card that is exhausted, then the Cash Value Used LED is lighted (5) and the Card Reader task is asked to eject the card (14). If the card is a credit card, then an authorization request is issued to the remote central facility via the Transmit Messages Queue (6). If a cash card was inserted that had cash value, or if a credit card was inserted and an authorization approval is received from the remote central facility (7), then the Pump Control task updates the transaction IHM (8). Of course, the remote central facility may refuse to authorize the credit transaction (7), in which case, the Pump Control task lights the Cannot Process Card LED (5) and the Card Reader task is asked to eject the card (14).

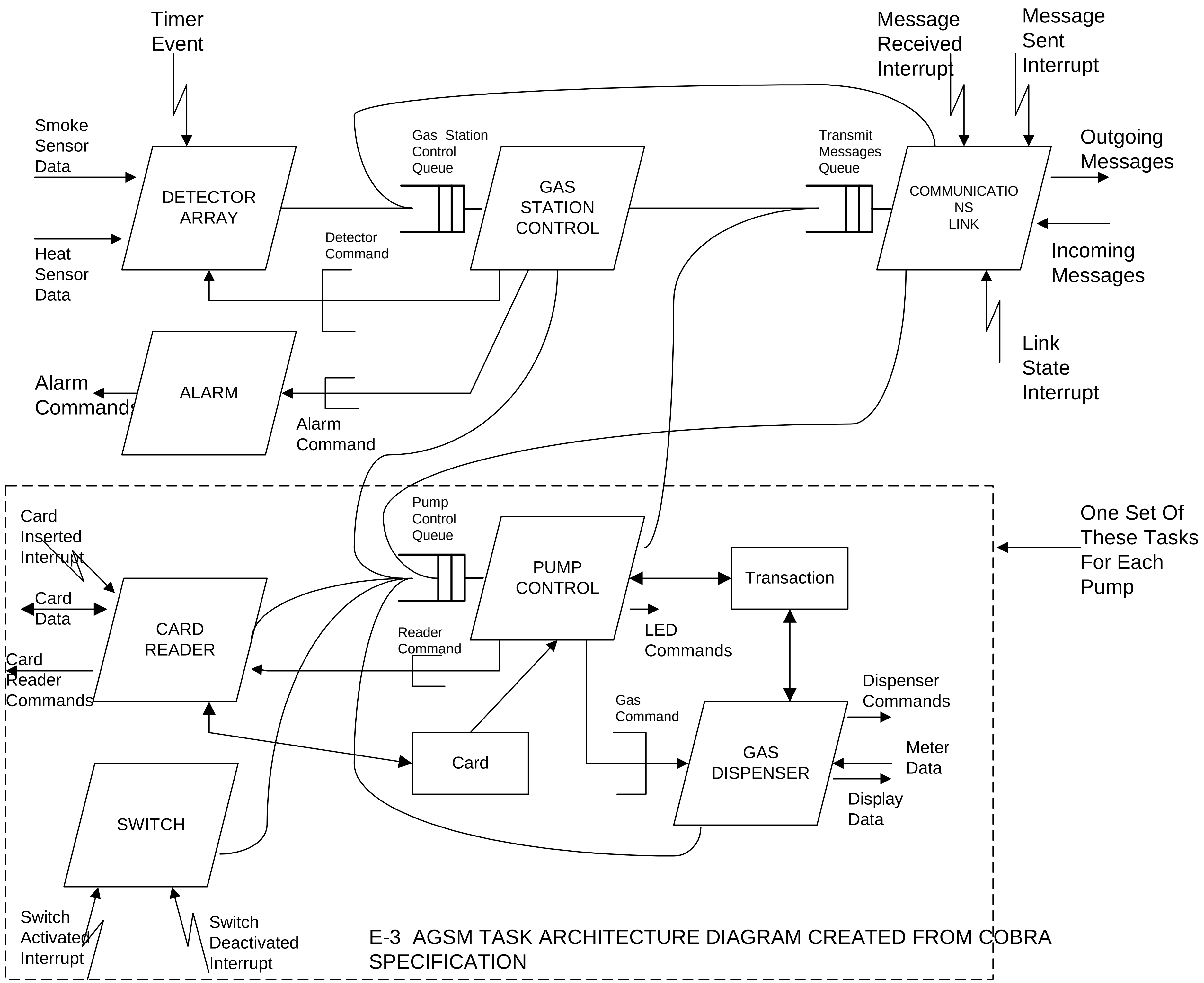
Once a transaction has been authorized, gas can only be dispensed if the pump switch is turned on by the customer. This event could occur in parallel with the transaction authorization or it could occur after the transaction authorization, but the switch must be on before gas can be dispensed. When the customer activates the switch (9), the Switch task generates a switch on event for the Pump Control Queue (10).

Once a transaction is authorized and the switch is on, the Pump Control task issues a start dispensing command (11) to the Gas Dispenser task. The Gas Dispenser task extracts any cash limit from the transaction object (12) and dispenses gas until the limit is reached, or until the Pump Control task orders a stop (11). Once complete, the Gas Dispenser task records the cost of gas purchased in the transaction object (12) and issues a stopped event to the Pump Control Queue (13).

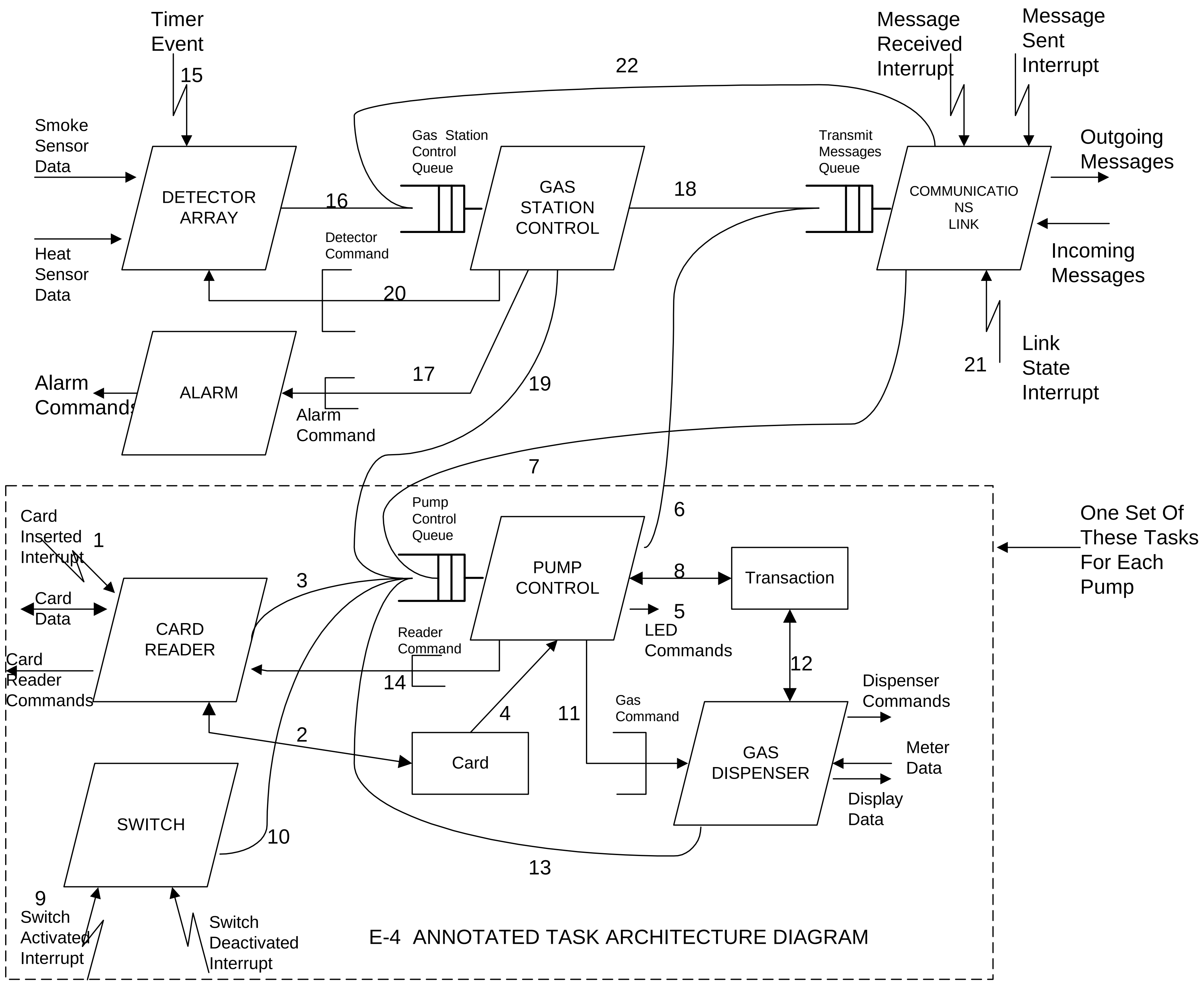
The Pump Control task then completes the transaction. For a cash transaction, a new cash value is computed for the customer's cash card and the card is updated (14). For a credit transaction, a credit transaction record is forwarded to the remote central facility via the Transmit Messages Queue (6). In either case, the customer's card will finally be ejected (14).

Now consider the flow of control activated by an emergency at the gas station. A timer event (15) will activate the Detectors task periodically so that the various smoke and heat detectors can be polled. Should a pre-defined threshold be exceeded at any of the detectors, the Detectors task will generate a threshold exceeded event for the Gas Station Control Queue (16). The Gas Station Control task, once alerted to the emergency, send an Alarm On Command (17) to the Alarm task and will send an alarm message to the remote central facility via the Transmit Messages Queue (18). The Gas Station Control task will also send a close command to each Pump Control Queue (19) and will disable the Detectors task (20). When the Pump Control task receives a close command any transaction in progress will be completed immediately and the pump will be closed.

The gas station can also be shutdown and restarted remotely from the central facility (22). When asked to shutdown, a close command is issued to each Pump Control Queue and when asked to restart an open command is issued to each Pump Control Queue (19). The gas station can also be shutdown and restarted due to changes in the state of the communications link. When the communications link state changes an interrupt is issued by the communications hardware (21). If the link goes down, then a link down event is issued to the Gas Station Control Queue; otherwise, a link up event is issued to the Gas Station Control Queue (22). If the link goes up, then the Gas Station Control task issues an open command to each Pump Control Queue (19). If the link goes down, then the Gas Station Control task issues a close command to each Pump Control Queue (19).



E-3 AGSM TASK ARCHITECTURE DIAGRAM CREATED FROM COBRA SPECIFICATION



E-4 ANNOTATED TASK ARCHITECTURE DIAGRAM

AUTOMATED GAS STATION MANAGER

COBRA/ADARTS

TASK BEHAVIOR SPECIFICATIONS

ALARM TASK BEHAVIOR SPECIFICATION

TASK: ALARM

A) TASK INTERFACE:

TASK INPUTS:

Messages: 1) Alarm Command (from Gas Station Control Task, tighly-coupled, no reply) - requests that the alarm be activated or deactivated.

parameters - action (0 = Turn_Off, 1 = Turn_On)

TASK OUTPUTS:

Data: 1) Alarm Commands - sounds the alarm.

IHMs REFERENCED:

NONE

B) TASK STRUCTURE:

Criterion: Asynchronous activation of a periodic output task.

Data Transformations: Alarm (3.3)

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by an Alarm Command message, and then executed periodically until deactivated.

D) PRIORITY:

Medium - outputs alarm frequency to sound alarm.

E) TASK EVENT SEQUENCING:

```
Loop A
        Await(Alarm Command)
        if Alarm Command.action = Turn_On
                then    Loop B
                                output ON to Alarm
                                if Waiting(Alarm Command)
                                        then if Alarm Command.action = Turn_Off
                                                then break Loop B
                                            fi
                                fi
                                delay 100 milliseconds
                        End Loop B
End Loop A
```

F) ERRORS DETECTED:

Ignores unrecognized actions and ignores redundant actions.

CARD READER TASK BEHAVIOR SPECIFICATION

TASK: Card Reader

A) TASK INTERFACE:

TASK INPUTS:

Events: 1) Card Inserted Interrupt (external event) - indicates customer inserted a card.

Messages: 1) Reader Command (from Pump Control Task, tightly-coupled, no reply) - requests that the Card Reader perform some action.

parameters - command (0 = Eject, 1 = write), new_cash_value (included only with write command).

Data: 1) Card Data - ASCII data encoding identificaton number and either a cash value (for a Cash Card) or and account number (for a Credit Card).

TASK OUTPUTS:

Messages: 1) Card Event (to Pump Control Queue, loosely-coupled) - indicates that a card was inserted.

parameters - action (0 = cash_card_inserted, 1 = credit_card_inserted), Card (reference to Card Object)

Data: 1) Card Reader Commands - include READ, WRITE, and EJECT.

2) Card Data - ASCII data encoding identificaton number and a new cash value.

IHMs REFERENCED:

CARD_READER - encapsulates card reader operations.

CARD - encapsulates information from a Cash or Credit Card (updated by task).

B) TASK STRUCTURE:

Criterion: Asynchronous Device I/O Dependency

Data Transformations: Card Reader (1.3)

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a card inserted interrupt or by a message arriving from the Pump Control Task.

D) PRIORITY:

High - for interrupt handling, Medium for responding to messages.

E) TASK EVENT SEQUENCING:

```
CARD INSERTED INTERRUPT VECTOR := wakeup(CARD_READER)

Loop
    Await()
End Loop

wakeup(CARD_READER)
    if (CARD_READER.read  = FALSE) then return
    Loop
        Await(Reader Command from Pump Control Task)
        if (Reader Command.command = Eject)
            then    CARD_READER.Eject
                    return
        elsif (Reader Command.command = write)
            then    CARD_READER.write(Reader Command.new_cash_value)
        fi
    End Loop
End_wakeup
```

F) ERRORS DETECTED:

Ignores unrecognized commands.

COMMUNICATIONS LINK TASK BEHAVIOR SPECIFICATION

TASK: Communications Link

A) TASK INTERFACE

TASK INPUTS:

Events:

1) Link State Interrupt (external event) - indicates that the communications hardware has detected a change in link state.

2) Message Received Interrupt (external event) - indicates that the communications hardware has completed reception of an incoming message.

3) Message Sent Interrupt (external event) - indicates that the communications hardware has completed transmission of an outgoing message.

Messages: 1) Transmit Messages Queue

a) Alarm Message (from Gas Station Control Task, loosely-coupled) - requests transmission of an Alarm Message to the Remote Central Facility.

parameters - remote_address, station_id

b) Authorization Request (from Pump Control Task, loosely-coupled) - requests transmission of an Authorization Request Message to the Remote Central Facility.

parameters - remote_address, station_id, pump_id, account_number

c) Credit Transaction (from Pump Control Task, loosely-coupled) - requests transmission of a Credit Transaction Message to the Remote Central Facility.

parameters - remote_address, station_id, pump_id, account_number, amount_purchased

d) Received (from self, loosely-coupled) - requests that an incoming message by decoded.

parameters - none

Data: 1) Incoming Messages from the communications hardware.

TASK OUTPUTS:

Messages: 1) RCF Event (to Gas Station Control Queue, loosely-coupled) - Remote Central Facility requests the gas station to change its operating state.

parameters - rcf_event (0 = shutdown, 1 = restart)

2) Link Event (to Gas Station Control Queue, loosely-coupled) - indicates that the communications link has changed state.

parameters - status (0 = link_down, 1 = link_up)

3) Credit Authorization Event (to Pump Control Queue, loosely-coupled) - informs the Pump about a customer's credit status.

parameters - status (0 = not_okay, 1 = okay)

Data: 1) Outgoing Messages to the communications hardware.

IHMs REFERENCED:

LINK - encapsulates operations for handling messages.

B) TASK STRUCTURE:

Criterion: Asynchronous Device I/O Dependency, Asynchronous Event Dependency, and Functional Cohesion

Data Transformations: Add to Rcv List (2.1), Decode Message Header (2.2), Analyze Link State (2.3), Add to Tx List (2.4), Transmit Message (2.5.1), Save Credit Transactions (2.5.2), Restore Credit Transactions (2.5.3)

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving in the Transmit Messages Queue and by the occurrence of various communications hardware interrupts.

D) PRIORITY:

High - for interrupt handling routines, Medium for processing the input queue.

E) TASK EVENT SEQUENCING

```
COMMUNICATIONS H/W INTERRUPT VECTOR 0 := Link_State
COMMUNICATIONS H/W INTERRUPT VECTOR 1 :=  Message_Rxed
COMMUNICATIONS H/W INTERRUPT VECTOR 2 :=  Message_Txed

Loop
    Await(Message Arrival in Transmit Messages Queue)
    Switch (Message Type)
        Case Alarm Message
        Case Authorization Request
        Case Credit Transaction
            LINK.add_to_tx_list(Message)
            LINK.transmit_message
            break
        Case Received
            decode_message(LINK)
            break
    End Switch
End Loop

Link_State
    event := input link_status_register
    Switch (LINK.analyze_link(event))
        Case 1
            send Link Event(status:=link_up) to Gas Station Control Queue
            break
        Case -1
            send Link Event(status:=link_down) to Gas Station Control Queue
            break
    End Switch
End_Link_State

Message_Rxed
    message := input received_message_pointer
    LINK.add_to_rcv_list(message)
    send Message(Type=Received) to Transmit Messages Queue
End_Message_Rxed

Message_Txed
    message := input sent_message_pointer
    free(message)
    LINK.transmit_message
End_Message_Txed
```

```
decode_message(LINK)

    Message := LINK.remove_from_rcv_list
    **Switch**(Message Type)
        **Case** Shutdown
            **send** RCF Event(rcf_event:=shutdown) **to** Gas Station Control Queue
            **break**
        **Case** Restart
            **send** RCF Event(rcf_event:=restart) **to** Gas Station Control Queue
            **break**
        **Case** Authorization Reply
            **Switch** response
                **Case** Not Authorized
                    **send** Credit Authorization Event(status:=not_okay)
                        **to** Pump Control Queue for Pump ID
                **Case** Authorized
                    **send** Credit Authorization Event(status:=okay)
                        **to** Pump Control Queue for Pump ID
            **End Switch**
    **End Switch**
    **free**(Message)
end_decode_message
```

F) ERRORS DETECTED:

Ignores unrecognized messages and events.

DETECTOR ARRAY TASK BEHAVIOR SPECIFICATION

TASK: Detector Array

A) TASK INTERFACE:

TASK INPUTS:

Events: 1) Timer (external event) - stimulates the task to poll its various smoke
and heat detectors.

Messages: 1) Detector Command (from Gas Station Task, tightly-coupled, no reply) - requests that the task enable or disable itself.

parameters - command (0 = disable, 1= enable)

Data: 1) Smoke Sensor Data from the smoke detectors.

2) Heat Sensor Data from the heat detectors.

TASK OUTPUTS:

Messages: 1) Emergency Event (to Gas Station Control Queue, loosely-coupled) - indicates that the gas station is on fire.

parameters - none

IHMs REFERENCED:

HEAT_DETECTOR - encapsulates raw sensor data from heat detectors and provides operations to test for threshold.

SMOKE_DETECTOR - encapsulates raw sensor data from smoke detectorsand provides operations to test for threshold.

B) TASK STRUCTURE:

Criterion: Periodic I/O Dependency

Data Transformations: Detector Array (3.2)

C) TIMING CHARACTERISTICS:

Activation: Periodic - activated by a timer, when task is enabled.

D) PRIORITY:

High - monitors safety conditions.

E) TASK EVENT SEQUENCING:

```
TIMER VECTOR :=  Poll_Detectors

Start Timer
Loop
        Await(Detector Command)
        Switch (Detector Command.command)
                Case disable
                        Stop Timer
                        break
                Case enable
                        Stop Timer
                        Start Timer
                        break
        End Switch
End  Loop

Poll_Detectors
        For Every HEAT_DETECTOR
                HEAT_DETECTOR.read
                if (HEAT_DETECTOR.monitor_sensor)
                        then send Emergency Event to Gas Station Control Queue
        End For
        For Every SMOKE_DETECTOR
                SMOKE_DETECTOR.read
                if (SMOKE_DETECTOR.monitor_sensor)
                        then send Emergency Event to Gas Station Control Queue
        End For
End_Poll_Detectors
```

F) ERRORS DETECTED:

Ignores unrecognized commands.

GAS DISPENSER TASK BEHAVIOR SPECIFICATION

TASK: Gas Dispenser

A) TASK INTERFACE:

TASK INPUTS:

Messages: 1) Gas Command (from Pump Control Task, tightly-coupled, no reply) - requests that the Gas Dispenser start or stop pumping gas.

parameters - command (0 = stop_dispensing, 1 = start_dispensing), Transaction (reference to transaction object).

Data: 1) Meter Data - count of the amount of gas dispensed.

2) TRANSACTION.limit - cash limit on amount of gas to be dispensed

TASK OUTPUTS:

Messages: 1) Stopped Event (to Pump Control Queue, loosely-coupled) - indicates that gas is no longer being dispensed.

Data: 1) Dispenser Commands - turns dispenser ON and OFF.

2) Display Data - amount and cost of gas to be displayed for customer viewing.

3) TRANSACTION.cost_of_gas - records the cost of the customer’s purchase.

IHMs REFERENCED:

GAS_DISPENSER - encapsulates dispenser operations.

TRANSACTION - encapsulates transaction data.

B) TASK STRUCTURE:

Criterion: Controls Gas Dispenser Device, Activated and Deactivated Asynchronously, but once activated, can run until a limit is reached.

Data Transformations: Gas Dispenser (1.5)

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving from the Pump Control Task.

D) PRIORITY:

High - must monitor dispensing operation on a polling basis.

E) TASK EVENT SEQUENCING:

```
Loop
    Await(Dispenser Command)
    if Dispenser Command.command not = start_dispensing
        then continue
    TRANSACTION := Dispenser Command.Transaction
    GAS_DISPENSER.clear_amount_dispensed
    GAS_DISPENSER.start_gas
    Loop
        GAS_DISPENSER.update_amount_dispensed
        GAS_DISPENSER.update_display
        if GAS_DISPENSER.?limit_reached(TRANSACTION)
            then break
        if Waiting (Dispenser Command)
            then if Dispenser Command.command = stop_dispensing
                then    break
    End Loop
    GAS_DISPENSER.stop_gas
    GAS_DISPENSER.update_amount_dispensed
    GAS_DISPENSER.update_display
    TRANSACTION.set_cost_of_gas(GAS_DISPENSER.get_amount_dispensed * 100 /
                                        GAS_DISPENSER.get_price)
    send Stopped Event to Pump Control Queue
End Loop
```

F) ERRORS DETECTED:

Ignores unrecognized commands.

GAS STATION CONTROL TASK BEHAVIOR SPECIFICATION

TASK: Gas Station Control

A) TASK INTERFACE:

TASK INPUTS:

Messages: 1) Gas Station Control Queue

a) RCF Event (from Communications Link Task, loosely-coupled) - indicates that a command from the Remote Central Facility has arrived.

parameters - rcf_event (0 = shutdown, 1 = restart)

b) Link Event (from the Communications Link Task, loosely-coupled) - indicates the communications link has changed state

parameters - status (0 = link_down, 1 = link_up)

c) Emergency Event (from Detectors Task, loosely-coupled) - indicates that the gas station is on fire.

parameters - none

TASK OUTPUTS:

Messages: 1) Gas Station Event (to Pump Control Queue, loosely-coupled) - requests the pump to change its operating state.

parameters - command (0 = close, 1 = open)

2) Alarm Message (to Transmit Messages Queue, loosely-coupled) - informs the Remote Central Facility about an emergency at the gas station.

parameters - remote address, station_id

3) Detector Command (to Detectors Task, tightly-coupled, no reply) - requests that the Detectors be enabled or disabled.

parameters - command (0 = disable, 1 = enable)

4) Alarm Command (to the Alarm Task, tightly-coupled, no reply) -
requests that the alarm be activated or deactivated

parameters - action (0 = Turn_Off, 1 = Turn_On)

IHMs REFERENCED:

GAS_STATION - encapsulates the Pump STD.

B) TASK STRUCTURE:

Criterion: Control Dependency (Gas Station task contains STD, encapuslated in the Gas Station.process_event operation, that sequences task operations.) and Sequential Cohesion.

Control Transformations: Gas Station Control (3.1)

Data Transformations: Send Alarm Message (3.4), Send Opens (3.5), and Send Closes (3.6)

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving in the Gas Station Control Queue.

D) PRIORITY:

Medium - lower than the I/O tasks.

E) TASK EVENT SEQUENCING:

```
Loop
    Await(Message Arrival in Gas Station Control Queue)
    Switch (Message Type)
        Case RCF Event
            if RCF Event.rcf_event = shutdown
                then event := SHUTDOWN
            elsif RCF Event.rcf_event = restart
                then event := RESTART
            fi
            break
        Case Link Event
            if Link Event.status = link_down
                then event := LINK_DOWN
```

```
                    elsif Link Event.status = link_up
                        then event := LINK_UP
                    fi
                    break
                Case Emergency Event
                    Emergency Event.event := THRESHOLD_EXCEEDED
                    break
        End Switch
        GAS_STATION.process_event(event)
End  Loop
```

F) ERRORS DETECTED:

Ignores Gas Station STD events that are not recognized.

PUMP CONTROL TASK BEHAVIOR SPECIFICATION

TASK: Pump Control

A) TASK INTERFACE:

TASK INPUTS:

Messages: 1) Pump Control Queue

a) Switch Event (from Switch Task, loosely-coupled) - indicates that a significant switch event has occurred

parameters - switch_event (0 = switch_off, 1 = switch_on)

b) Card Event (from Card Reader Task, loosely-coupled) - indicates that a card was inserted.

parameters - action (0 = cash_card_inserted, 1 = credit_card_inserted), Card (reference to Card Object)

c) Gas Station Event (from Gas Station Control Task, loosely-coupled) - requests that the Pump change its operating state.

parameters - command (0 = close, 1 = open)

d) Authorization Event (from the Communications Link Task or from internal to Pump Control Task, loosely-coupled)

parameters - status (0 = not_okay, 1 = okay)

e) Stopped Event (from Gas Dispenser Task, loosely-coupled) - indicates that gas is no longer being dispensed.

parameters - none

Data: 1) Cash and Credit Card Information from the Card IHM (which is created by the Card Reader Task)

2) Transaction Information from the Transaction IHM (which is created by this task)

TASK OUTPUTS:

Messages: 1) Reader Command (to Card Reader Task, tightly-coupled, no reply) - requests the Card Reader to execute a command.

parameters - command (0 =Eject, 1 = write), new_cash_value (included only with the write command)

2) Gas Command (to Gas Dispenser Task, tightly-coupled, no reply) - requests that the Gas Dispenser start or stop pumping gas.

parameters - command (0 = stop_dispensing, 1 = start_dispensing), Transaction (reference to transaction object).

3) Authorization Request (to the Transmit Messages Queue, loosely-coupled) - requests that the Remote Central Facility check credit authorization for a customer.

parameters - remote_address, station_id, pump_id, account_number

4) Credit Transaction (to the Transmit Messages Queue, loosely-coupled) - informs the Remote Central Facility about a completed credit transaction.

parameters - remote_address, station_id, pump_id, account_number, amount_purchased

Data: 1) Transaction - created by this task.

2) LED Commands - to light and extinguish the Pump’s LEDs

IHMs REFERENCED:

PUMP - encapsulates the Pump STD

CARD - encapsulates cash or credit card information read by the Card Reader Task.

TRANSACTION - encapsulates an on-going transaction.

B) TASK STRUCTURE:

Criterion: Control Dependency (Pump task contains STD, encapuslated in the Pump.process_event operation, that sequences task operations) and Sequential Cohesion.

Control Transformation: Control Pump (1.1.1)

Data Transformations: Authorize Transaction (1.1.2), Establish Transaction (1.1.3),
Complete Transaction (1.1.4), Reject Transaction (1.1.5)

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving in the Pump Control Queue.

D) PRIORITY:

Medium - lower than the I/O tasks.

E) TASK EVENT SEQUENCING:

```
Loop
    Await(Message Arrival in Pump Control Queue)
    Switch (Message Type)
        Case Switch Event
            if Switch Event.switch_event = switch_off
                then event :=  OFF
            elsif Switch Event.switch_event = switch_on
                then event := ON
            fi
            break
        Case Card Event
            CARD := Card Event.Card
            if Card Event.action = cash_card_inserted
                then    event := CASH_CARD_INSERTED
            elsif Card Event.action = credit_card_inserted
                then    event := CREDIT_CARD_INSERTED
            fi
            break
        Case Gas Station Event
            if Gas Station Event.command = close
                then event := CLOSE
            elsif Gas Station Event.command = open
                then event := OPEN
            fi
            break
        Case Credit Authorization Event
            if Credit Authorization Event.status = not_okay
                then event := NOT_AUTHORIZED
            elsif status = okay
                then event := AUTHORIZED
            fi
            break
        Case Stopped Event
            event := Stopped
            break
    End Switch
    PUMP.process_event(event, CARD)
End  Loop
```

F) ERRORS DETECTED:

Ignores Pump STD events that are not recognized.

SWITCH TASK BEHAVIOR SPECIFICATION

TASK: Switch

A) TASK INTERFACE:

TASK INPUTS:

Events: 1) Switch Activated Interrupt (external event) - indicates pump switch was turned on.
2) Switch Deactivated Interrupt (external event) - indicates pump switch was turned off.

TASK OUTPUTS:

Messages: 1) Switch Event (to Pump Control Queue, loosely-coupled) - indicates that a switch event has occurred.

parameters - switch_event (0 = switch_off, 1 = switch_on)

IHMs REFERENCED:

None

B) TASK STRUCTURE:

Criterion: Asynchronous Device I/O

Data Transformations : Switch (1.2)

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a Switch Interrupt.

D) PRIORITY:

High - captures and records hardware interrupts that would otherwise be lost.

E) TASK EVENT SEQUENCING:

```
SWITCH INTERRUPT VECTOR := handle_switch_interrupt()
Loop
    Await()
End Loop

handle_switch_interrupt()
    input SWITCH_INTERRUPT_REGISTER
    if switch_activated
        then send Switch Event(switch_event:=switch_on)
            to Pump Control Queue
    fi
    if switch_deactivated
        then send Switch Event(switch_event:=switch_off)
            to Pump Control Queue
    fi
```

F) ERRORS DETECTED:

None.

AUTOMATED GAS STATION MANAGER

COBRA/ADARTS

INFORMATION HIDING MODULE

SPECIFICATIONS

AUTOMATED GAS STATION MANAGER INFORMATION HIDING MODULE SPECIFICATIONS

IHM: CARD_READER

Information Hidden: The details of reading and writing information on a magnetic card strip and

of ejecting the card from a specific card reader.

Module Structure: Device Interface Module

Assumptions: Will only be accessed sequentially, i.e., no concurrent access.

Anticipated Changes:

Operations:

1) read

Function: Reads the first field, i.e., identification number, on a card magnetic strip to establish whether the card is a credit or cash card. Reads the remaining information on the card (i.e., cash value for a cash card and account number for a credit card), converts the ASCII data from the card into numeric form and stores it using the CARD IHM. Finally, returns TRUE. If the identification number cannot be recognized, then the card is ejected and FALSE is returned.

Input Parameters: None

Output Parameters: Status (FALSE = invalid card, TRUE = valid card)

2) eject

Function: Sends an Eject Command to the Card Reader.

Input Parameters: None

Output Parameters:

3) write

Function: Writes a new cash value to a cash card.

Input paramters: new cash value

Output parameters: None

IHM: CARD

Information Hidden: Card Information Data Store

Module Structure: Data Abstraction Module

Assumptions: Shared between two tasks; however, access is sequenced so that no simultaneous access occurs.

Anticipated Changes:

Operations:

1) set_id_number

Function: Stores input id into identification number. Sets cash value and account number to zero.

Input Parameters: id

Output Parameters: None

2) get_id_number

Function: Extracts internal identification number and returns it in output parameter.

Input Parameters: None

Output Parameters: identification number

3) is_cash_card

Function: Determines if the CARD is holding cash card or credit card information.

Input Parameters: None

Output Parameters: TRUE if Cash Card, FASLE if Credit Card

4) set_cash_value

Function: If this is a Cash Card, then stores input value into cash value.

Input Parameters: value

Output Parameters: None

5) get_cash_value

Function: If this is a Cash Card, then extracts internal cash value and returns it in output parameter.

Input Parameters: None

Output Parameters: cash value

6) set_account_number

Function: If CARD is credit_card, then stores input number into account number.

Input Parameters: number

Output Parameters: None

7) get_account_number

Function: If CARD is credit card, then extracts internal account number and returns it in output parameter.

Input Parameters: None

Output Parameters: account number

IHM: GAS_DISPENSER

Information Hidden: Details of the Gas Dispenser hardware

Module Structure: Device Interface Module

Assumptions: Accessed by one task only.

Anticipated Changes:

Operations:

1) clear_amount_dispensed

Function: Blanks to GALLONS display and the COST display. Sets the amount dispensed attribute to zero.

Input Parameters: None

Output Parmeters: None

2) update_amount_dispensed

Function: Reads the current value of the gas dispenser's meter, converts the value to an integer and stores the integer as the amount dispensed attribute.

Input Parameters: None

Output Parameters: None

3) update_display

Function: Converts amount dispensed attribute into a display format and computes cost of gas dispensed as a display format and outputs these values to the appropriate displays.

Input Parameters: None

Output Parameters: None

4) get_amount_dispensed

Function: Returns the amount dispensed attribute

Input Parametes: None

Output Parameters: amount dispensed

5) get_price

Function: Returns the price per gallon

Input Parameters: None

Output Parameters: price per gallon

6) ?limit_reached

Function: Determines if the cost limit of gas to be dispensed has been reached.

Input Parameters: TRANSACTION

Output Parameters: TRUE (if limit reached) or FALSE (if limit not reached)

7) start_gas

Function: Outputs a START COMMAND to the gas dispenser

Input Parameters: None

Output Parameters: None

8) stop_gas

Function: Outputs a STOP COMMAND to the gas dispenser.

Input Parameters: None

Output Parameters: None

IHM: GAS_STATION

Information Hidden: Gas Station Control State Transition Diagram and a few supporting operations.

Module Structure: State Transition Module (primarily)

Assumptions: Used by only one task

Anticipated Changes:

Operations:

1) process_event

Function: Execute the Gas Station Control STD.

Input Parameters: Event

Output Paramters: None

IHM: HEAT_DETECTOR

Information Hidden: Details of accessing the heat detector hardware.

Module Structure: Device Interface Module

Assumptions: Will be accessed by only one task.

Anticipated Changes:

Operations:

1) read

Function: Inputs sensor data from a heat detector and converts that sensor data to an internal form that is stored in the processed sensor data attribute.

Input Parameters: None

Output Paramters: None

2) monitor_sensor

Function: Computes a delta between processed sensor data and a known threshold If the delta exceeds the threshold, then returns TRUE; else, returns FALSE.

Input Parameters: None

Output Parameters: TRUE if threshold exceeded, FALSE otherwise.

IHM: LINK

Information Hidden: Details of the communications link hardware.

Module Structure: Device Interface Module

Assumptions: Accessed by only one task.

Anticipated Changes:

Operations:

1) add_to_rcv_list

Function: Inserts an incoming message at the end of the receive list

Input Parameters: Message to insert

Output Parameters: None

2) analyze_link_state

Function: Determines whether the link has gone up or down. Returns new state if change occurs, FALSE otherwise.

Input Parameter: Event

Output Parameter: 1 = Link came up, -1 = Link went down, 0 = no change.

3) add_to_tx_list

Function: Inserts the input message a the end of the transmit list, then stimulates a transmission.

Input Parameters: Message to send

Output Parameters: None

4) transmit message

Function: If the transmit list is not empty and the link is up and a message is not already being transmitted, then the next message transmission is started. If the link is down, then save_credit_transactions is called.

Input Parameters: None

Output Parameters: None

5) save_credit_transactions

Function: Cycles through the transmit message list: if a message is a credit transaction, then it is written to the end of a file. In any case, the message is removed from the transmit message list.

Input Parameters: None

Output Parameters: None

6) restore_credit_transactions

Function: Cycles through the file of saved credit transactions, moving each credit transaction from the file to the head of the transmit message list. The file is then deleted.

Input Parameters: None

Output Parameters: None

7) remove_from_rcv_list

Function: Removes a message, if any, from the head of the receive message list.

Input Parameters: None

Output Parameters: pointer to message or zero, if the receive message list is empty.

IHM: PUMP

Information Hidden: The Pump Control STD

Module Structure: State Transition Module

Assumptions: Accessed by a single task.

Anticipated Changes:

Operations:

1) process_event

Function: Execute the appropriate transition in the Pump Control STD.

Input Parameters: Event and CASH_CARD or CREDIT_CARD

Output Parameters: None

IHM: SMOKE_DETECTOR

Information Hidden: Details of accessing the smoke detector hardware.

Module Structure: Device Interface Module

Assumptions: Will be accessed by only one task.

Anticipated Changes:

Operations:

1) read

Function: Inputs sensor data from a smoke detector and converts that sensor data to an internal form that is stored in the processed sensor data attribute.

Input Parameters: None

Output Paramters: None

2) monitor_sensor

Function: Computes a delta between processed sensor data and a known threshold If the delta exceeds the threshold, then returns TRUE; else, returns FALSE.

Input Parameters: None

Output Parameters: TRUE if threshold exceeded, FALSE otherwise.

IHM: TRANSACTION

Information Hidden: On-going Transaction Data Store

Module Structure: Data Abstraction Module

Assumptions: Access shared by two tasks, but access is sequenced so that no simultaneous access will occur.

Anticipated Changes:

Operations:

1) get_limit

Function: Extracts the value stored in the limit attribute.

Input Parameters: None

Output Parameters: limit

2) set_cost_of_gas

Function: Stores input value into the cost of gas attribute.

Input Parameters: value

Output Parameters: None

3) is_cash_transaction

Function: Determines if transaction is cash or credit.

Input Paramters: None

Output Parameters: TRUE is cash transaction, FALSE is credit transaction.

4) set_limit

Function: Sets the limit to the input value.

Input Parameters: value

Output Parameters: None

5) get_cost_of_gas

Function: Returns the value of the cost of gas field.

Input Parameters: None

Output Parameters: cost of gas

AUTOMATED GAS STATION MANAGER

COBRA/ADARTS

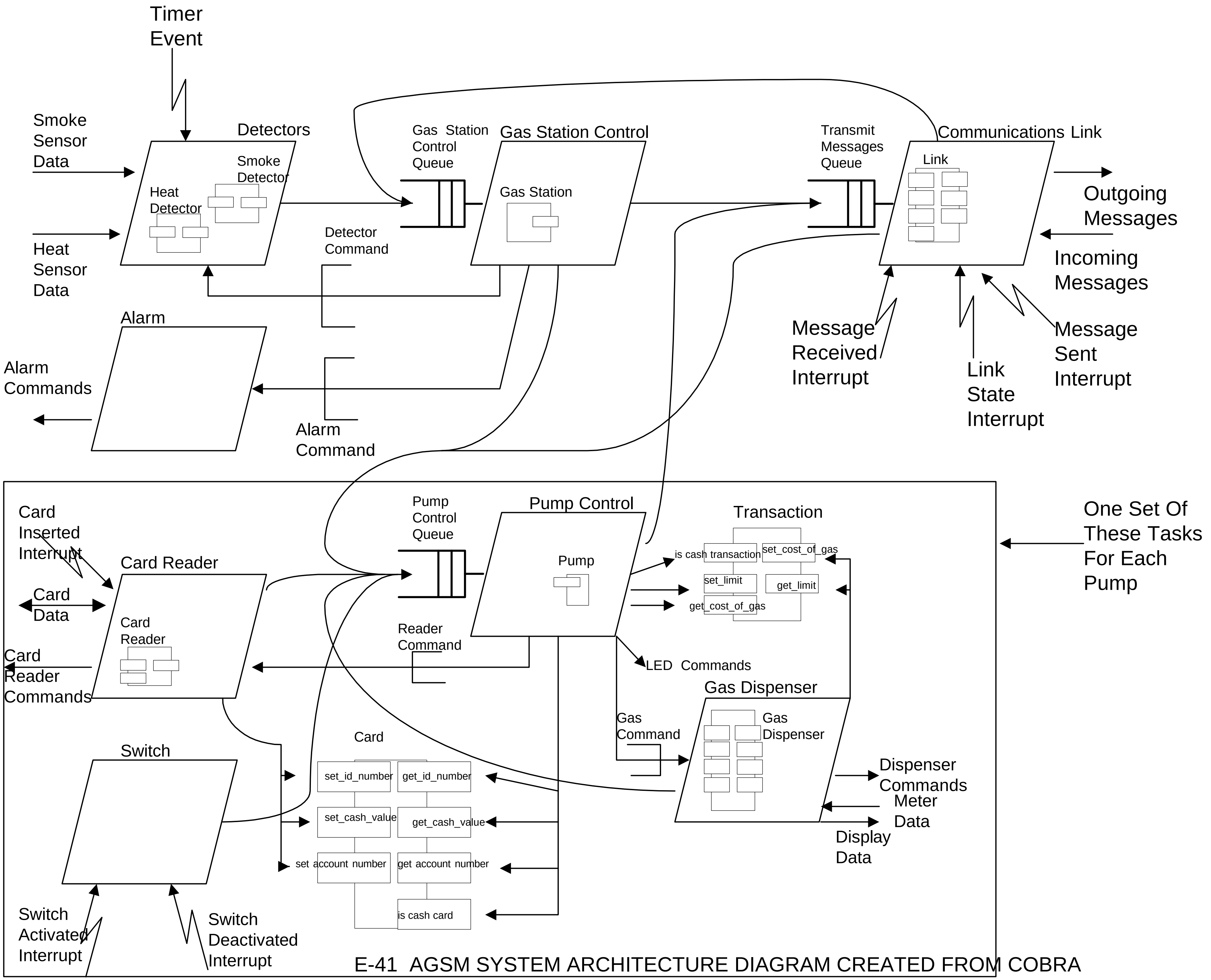
SYSTEM ARCHITECTURE

Overview Of AGMS System Architecture

The AGMS system architecture, illustrated in Figure E-41, augments the task architecture, shown previously in Figure E-3, by identifying the information hiding modules (IHMs) within the system and by showing the allocation of those IHMs among tasks. A Gas Station IHM is included within the Gas Station Control task. Within the Detectors task, a Heat Detector IHM is included for each heat detector in the gas station and a Smoke Detector IHM is included for each smoke detector. The Communications Link task includes a Link IHM.

Turning to the tasks associated with each pump, the Pump Control task includes a Pump IHM, the Card Reader task includes a Card Reader IHM, and the Gas Dispenser task includes a Gas Dispenser IHM. (The Switch task contains and uses no IHMs.) Within each set of pump tasks, the Card and Transaction IHMs play a key role.

The Card IHM is used by the Card Reader task to store the information read from a customer's cash card or credit card. This information is them accessible to the Pump Control Task. The Pump Control Task uses the information maintained by the Card IHM to authorize a cash transaction and to compute a new cash value. The Pump Control Task uses the information maintained by the Card IHM to request, from the remote central facility, authorization of a credit transaction and to report the completion of a transaction to the remote central facility. The Gas Dispenser task uses the Transaction IHM to determine if a limit is set on the cost of gas to be dispensed and to store the cost of any gas that is dispensed from the gas dispenser. The Pump Control task uses the Transaction IHM to set a transaction limit, to get the cost of the gas dispensed, and to determine if the transaction is cash or credit.



E-41 AGSM SYSTEM ARCHITECTURE DIAGRAM CREATED FROM COBRA SPECIFICATION

APPENDIX F. AUTOMATED GAS STATION MANAGER DESIGN FROM OMT SPECIFICATION USING OODARTS

AUTOMATED GAS STATION MANAGER

OMT/OODARTS

OVERVIEW OF

ACTIVE OBJECT ARCHITECTURE

Overview Of AGSM Active Object Architecture

The Automated Gas Station Management (AGSM) software comprises a set of three active objects (AOs) to manage the gas station and a set of five active objects (AOs) for each pump. This architecture is illustrated in Figure F-3. The architecture is reproduced in Figure F-4 with the inter-object communications identified by numbering the arcs between the AOs.

As shown in Figure F-3, each gas station consists of three AOs: 1) Gas Station Control, 2) Detectors, and 3) Communications Link. The Gas Station Control AO, activated by the arrival of messages in the Gas Station Control Queue, manages the operation of the gas station software. The Detectors AO, activated by a periodic timer event, polls each of the smoke and heat detectors installed at the gas station. When the concentration of smoke particles or the temperature exceeds a pre-defined threshold at any of the smoke or heat detectors, respectively, the Detectors AO notifies the Gas Station Control AO. The Communications Link AO acts as a conduit between the gas station and a remote central facility. Messages sent from the station to the remote central facility and messages received at the station from the remote cental facility pass through the Communications Link AO. The AO is activated by messages arriving in the Transmit Messages Queue and by any of four hardware interrupts (message sent, message received, link state change, and time-out).

The gas station includes a number of pumps, each of which is controlled through five AOs, as shown in Figure F-3. The main object managing a pump is the Pump Control AO which sequences and synchronizes external inputs that affect the pump. The Pump Control AO is activated by the arrival of a message in the Pump Control Queue. The Gas Dispenser Control Ao, activated by a message from the Pump Control AO, controls the dispensing of gasoline through the pump's gas dispenser. The Card Reader Control Ao, when activated by a Card Inserted Interrupt, extracts information from a customer's cash or credit card, creates an appropriate Card data abstraction object, and awaits further commands issued under control of the Pump Control AO. The Switch Monitoring AO simply converts switch activation and deactivation interrupts into internal events and sends them to the Pump Control Queue. The LED Control AO, activated by messages arriving in the Light Queue, controls the lighting and extinguishing of the two LEDs (Cash Value Used and Cannot Process Card) on the pump.

Turning attention to Figure F-4, some typical flows of control through the system can be described. A customer approaches an operating pump at an open gas station and inserts a credit or cash card causing a Card Inserted Interrupt (1). The Card Reader Control Ao determines the type of card that was inserted. If the card is not recognized, then the LED Control AO is asked to light the Cannot Process Card LED (2). If the card is recognized, an appropriate data abstraction object is created and the information from the card is stored within the object (3). A card inserted event is then passed to the Pump Control Queue (4). The Pump Control AO examines the card inserted event and related card information (5). If the card is a cash card that is exhausted, then a request is sent to the LED Control AO to light the Cash Value Used LED (6) and the Card Reader Control AO is asked to eject the card (14). If the card is a credit card, then an authorization request is issued to the remote central facility via the Transmit Messages Queue (7). If a cash card was inserted that had cash value, or if a credit card was inserted and an authorization approval is received from the remote central facility (8), then the Pump Control AO creates an appropriate transaction object (9). Of course, the remote central facility may refuse to authorize the credit transaction (8), in which case, the Pump Control AO issues a request to the LED Control AO to light the Cannot Process Card LED (6) and the Card Reader Control Ao is asked to eject the card (14).

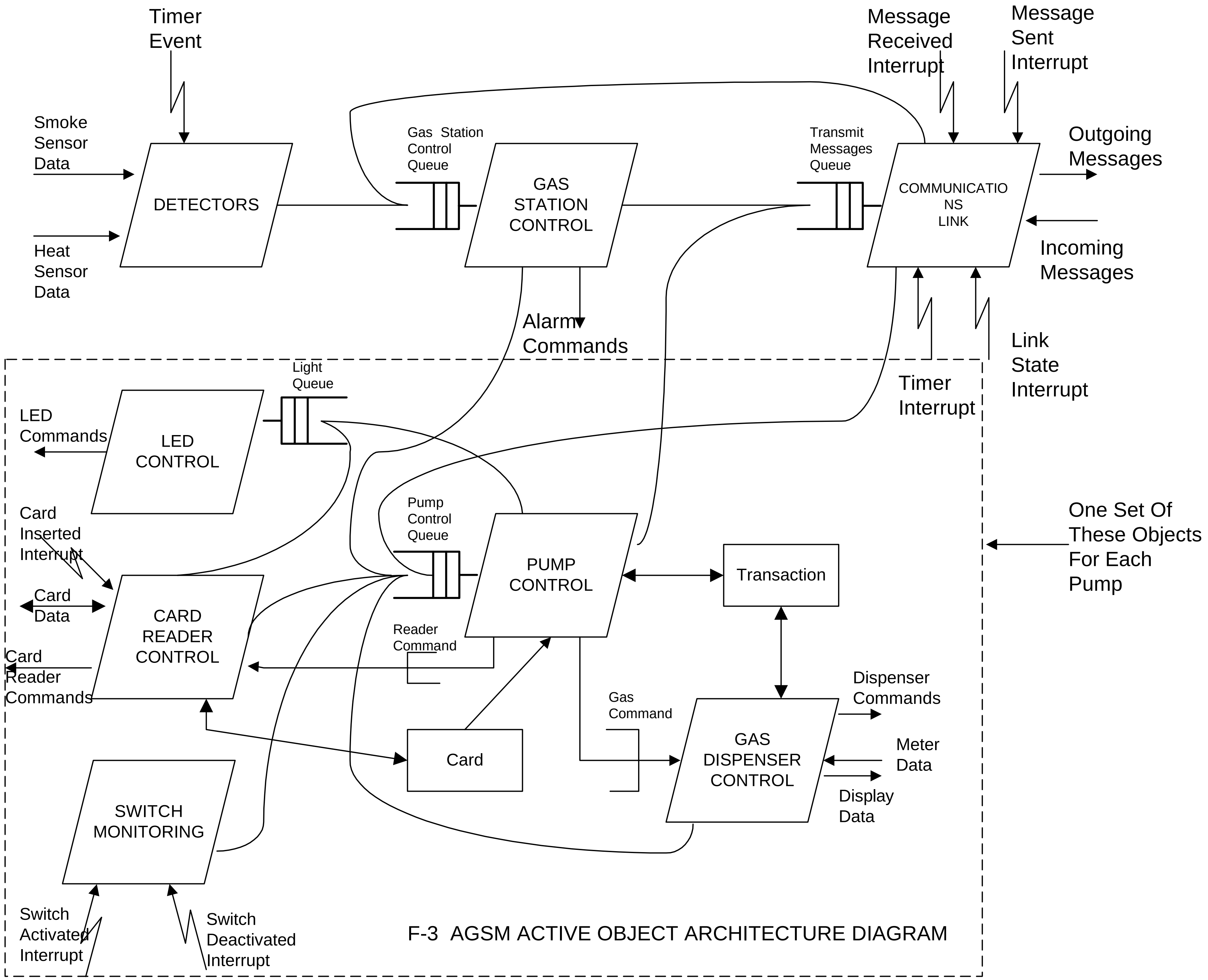
Once a transaction has been authorized, gas can only be dispensed if the pump switch is turned on by the customer. This event could occur in parallel with the transaction authorization or it could occur after the transaction authorization, but the switch must be on before gas can be dispensed. When the customer activates the switch (10), the Switch Monitoring Ao generates a switch on event for the Pump Control Queue (10a).

Once a transaction is authorized and the switch is on, the Pump Control AO issues a start dispensing command (11) to the Gas Dispenser Control Ao. The Gas Dispenser Control Ao extracts any cash limit from the transaction object (12) and dispenses gas until the limit is reached, or until the Pump Control AO orders a stop (11). Once complete, the Gas Dispenser Control Ao records the cost of gas purchased in the transaction object (12) and issues a stopped event to the Pump Control Queue (13).

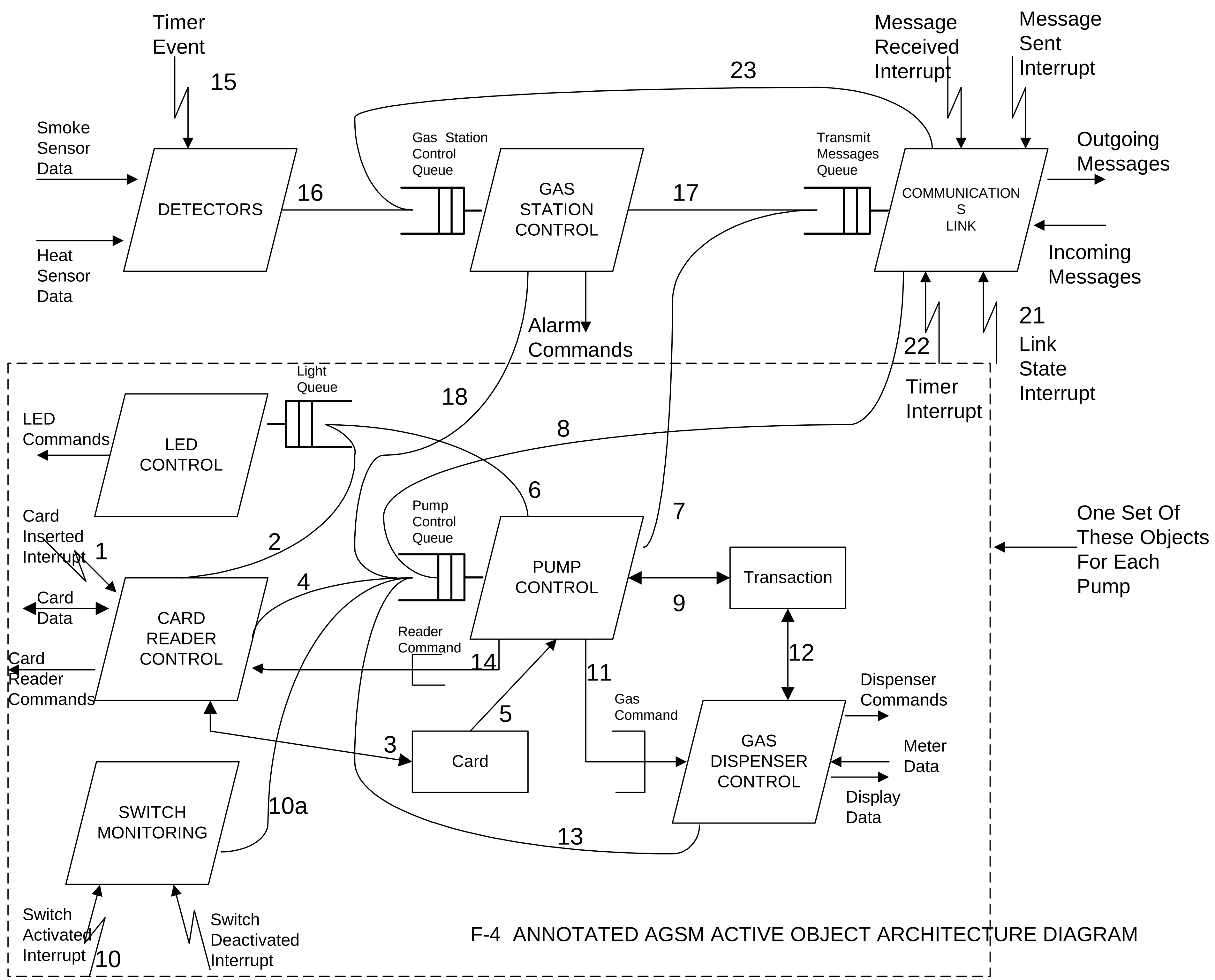
The Pump Control AO then completes the transaction. For a cash transaction, a new cash value is computed for the customer's cash card and the card is updated (14). For a credit transaction, a credit transaction record is forwarded to the remote central facility via the Transmit Messages Queue (7). In either case, the customer's card will finally be ejected (14).

Now consider the flow of control activated by an emergency at the gas station. A timer event (15) will activate the Detectors AO periodically so that the various smoke and heat detectors can be polled. Should a pre-defined threshold be exceeded at any of the detectors, the Detectors AO will generate a threshold exceeded event for the Gas Station Control Queue (16). The Gas Station Control AO, once alerted to the emergency, will sound the gas station alarm and will send an alarm message to the remote central facility via the Transmit Messages Queue (17). The Gas Station Control AO will also send a close command to each Pump Control Queue (18). When the Pump Control AO receives a close command any transaction in progress will be completed immediately and the pump will be closed.

The gas station can also be shutdown and restarted remotely from the central facility (23). When asked to shutdown, a close command is issued to each Pump Control Queue and when asked to restart an open command is issued to each Pump Control Queue (18). The gas station can also be shutdown and restarted due to changes in the state of the communications link. When the communications link state changes an interrupt is issued by the communications hardware (21). The Communications Link AO will then start a timer. If the link state doesn't change for the duration of the time period (22), then the link state will be changed. If the link goes down, then a link down event is issued to the Gas Station Control Queue; otherwise, a link up event is issued to the Gas Station Control Queue (23). If the link goes up, then the Gas Station Control AO issues an open command to each Pump Control Queue (18). If the link goes down, then the Gas Station Control AO issues a close command to each Pump Control Queue (18).



F-3 AGSM ACTIVE OBJECT ARCHITECTURE DIAGRAM



F-4 ANNOTATED AGSM ACTIVE OBJECT ARCHITECTURE DIAGRAM

AUTOMATED GAS STATION MANAGER

OMT/OODARTS

ACTIVE OBJECT

BEHAVIOR SPECIFICATIONS

CARD READER CONTROL ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: Card Reader Control

A) ACTIVE OBJECT INTERFACE:

AO INPUTS:

Events: 1) Card Inserted Interrupt (external event) - indicates customer inserted a card.

Messages: 1) Reader Command (from Pump Control AO, tightly-coupled, no reply) - requests that the Card Reader perform some action.

parameters - command (0 = Eject, 1 = write), new_cash_value (included only with write command).

Data: 1) Card Data - ASCII data encoding identificaton number and either a cash value (for a Cash Card) or and account number (for a Credit Card).

AO OUTPUTS:

Messages: 1) Card Event (to Pump Control Queue, loosely-coupled) - indicates that a card was inserted.

parameters - action (0 = cash_card_inserted, 1 = credit_card_inserted), Card (reference to Card Object)

2) Light Command (to Light Queue, loosely-coupled) - request that an LED be lighted.

parameters - LED_number (0 = Cash_Value_Used, 1 = Cannot_Process_Card)

Data: 1) Card Reader Commands - include READ, WRITE, and EJECT.

2) Card Data - ASCII data encoding identificaton number and a new cash value.

OBJECT CLASSES REFERENCED:

CARD_READER - encapsulates card reader device operations.

CASH_CARD - encapsulates information from a Cash Card .
(created by Card Reader)

CREDIT_CARD - encapsulates information from a Corporate Credit Card
(created by Card Reader).

B) ACTIVE OBJECT STRUCTURE:

Criterion: Asynchronous Device I/O Dependency and Functional Cohesion.

Objects : Card Reader

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a card inserted interrupt or by a message arriving from the Pump Control AO or from a Transaction Object executing within the Pump Control AO's thread of control.

D) PRIORITY:

High - for interrupt handling, Medium for responding to messages.

E) ACTIVE OBJECT EVENT SEQUENCING:

```
Card_Reader:CARD_READER
Pump_Control:ACTIVE_OBJECT
Reader_Command:MESSAGE
Card_Inserted:AO_INTERRUPT

Create(pc:ACTIVE_OBJECT, pcq:ACTIVE_QUEUE, lq:ACTIVE_QUEUE)
    Card_Reader.Create(pcq, lq, CR_REGISTER_BASE)
    Card_Inserted.Create(CI_INTERRUPT_VECTOR,wakeup)
    Pump_Control := pc
    Reader_Command.Create(READER_COMMAND)
end_Create

execute()
    Loop
        Await()
    End Loop
end_execute

wakeup()
    if (Card_Reader.read = FALSE) then return
    Loop
```

```
                Wait for (Reader_Command) from Pump_Control
                if (Reader_Command.command = Eject)
                        then    Card_Reader.Eject
                                return
                elsif (Reader_Command.command = write)
                        then    Card_Reader.write(Reader_Command.new_cash_value)
                fi
        End Loop
End_wakeup
```

F) ERRORS DETECTED:

Ignores unrecognized commands.

COMMUNICATIONS LINK ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: Communications Link

A) ACTIVE OBJECT INTERFACE

AO INPUTS:

Events: 1) Link State Interrupt (external event) - indicates that the communications hardware has detected a change in link state.

2) Message Received Interrupt (external event) - indicates that the communications hardware has completed reception of an incoming message.

3) Message Sent Interrupt (external event) - indicates that the communications hardware has completed transmission of an outgoing message.

4) Time Out Interrupt (external event) - indicates expiration of a timer provided by the communications hardware.

Messages: 1) Transmit Messages Queue

a) Alarm Message (from Gas Station Control AO, loosely-coupled) - requests transmission of an Alarm Message to the Remote Central Facility.

parameters - remote_address, station_id

b) Authorization Request (from Pump Control AO, loosely-coupled) - requests transmission of an Authorization Request Message to the Remote Central Facility.

parameters - remote_address, station_id, pump_id, account_number

c) Credit Transaction (from Pump Control AO, loosely-coupled) - requests transmission of a Credit Transaction Message to the Remote Central Facility.

parameters - remote_address, station_id, pump_id, account_number, amount_purchased

d) Received (from self, loosely-coupled) - requests that an incoming message by decoded.

parameters - none

Data: 1) Incoming Messages from the communications hardware.

AO OUTPUTS:

Messages: 1) RCF Event (to Gas Station Control Queue, loosely-coupled) - Remote Central Facility requests the gas station to change its operating state.

parameters - rcf_event (0 = shutdown, 1 = restart)

2) Link Event (to Gas Station Control Queue, loosely-coupled) - indicates that the communications link has changed state.

parameters - status (0 = link_down, 1 = link_up)

3) Credit Authorization Event (to Pump Control Queue, loosely-coupled) - informs the Pump about a customer’s credit status.

parameters - status (0 = not_okay, 1 = okay)

Data: 1) Outgoing Messages to the communications hardware.

OBJECT CLASSES REFERENCED:

LINK - encapsulates the Link STD and provides some operations for handling messages.

B) ACTIVE STRUCTURE:

Criterion: Asynchronous Device I/O Dependency and Control Dependency (Link object encapsulates an STD in the analyze_link operation), and Functional Cohesion (Link object contains operations to support the communications processing).

Objects: Link

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving in the Transmit Messages Queue and by the occurrence of various communications hardware interrupts.

D) PRIORITY:

High - for interrupt handling routines, Medium for processing the input queue.

E) ACTIVE OBJECT EVENT SEQUENCING

Link**:LINK**

Transmit_Messages_Queue**:ACTIVE_QUEUE**

Link_State_Interrupt**:AO_INTERRUPT**
RX_Interrupt**:AO_INTERRUPT**
TX_Interrupt**:AO_INTERRUPT**
Time_Out_Interrupt**:AO_INTERRUPT**

```
Create(tmq:ACTIVE_QUEUE, gscq:ACTIVE_QUEUE, pcql:ARRAY[ACTIVE_QUEUE])
    Link.Create(gscq, pcql, LINK_COMMAND_REGISTER)
    Transmit_Messages_Queue := tmq
    Link_State_Interrupt.Create(LS_INTERRUPT_VECTOR, Link_State)
    RX_Interrupt.Create(RX_INTERRUPT_VECTOR, Message_Rxed)
    TX_Interrupt.Create(TX_INTERRUPT_VECTOR, Message_Txed)
    Time_Out_Interrupt.Create(TO_INTERRUPT_VECTOR, Time_Out)
end_Create

execute()
    message:MESSAGE

    message.Create(SELECTOR)
    Loop
        Wait for message in Transmit_Messages_Queue
        Switch (message.type)
            Case Alarm Message
            Case Authorization Request
            Case Credit Transaction
                Link.add_to_tx_list(message)
                Link.transmit_message
                break
            Case Received
                Link.decode_message
                break
        End Switch
    End Loop
end_execute

Link_State
```

```
        event:**LINK_EVENT**

        event := **input** LINK_STATUS_REGISTER
        Link.analyze_link(event)
End_Link_State

Message_Rxed
        message:**MESSAGE**
        received:**MESSAGE**

        message.Create(SELECTOR)
        message := **input** RECEIVED_MESSAGE_POINTER
        Link.add_to_rcv_list(message)
        received.Create(RECEIVED)
        **send** received **to** Transmit_Messages_Queue
End_Message_Rxed

Message_Txed
        message:**MESSAGE**

        message := **input** SENT_MESSAGE_POINTER
        **free**(message)
        Link.transmit_message
End_Message_Txed


Time_Out
        Link.analyze_link(TIME_OUT)
End_Time_Out
```

F) ERRORS DETECTED:

Ignores unrecognized messages and events.

DETECTORS ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: Detectors

A) ACTIVE OBJECT INTERFACE:

AO INPUTS:

Events: 1) Timer (external event) - stimulates the task to poll its various smoke and heat detectors.

Data: 1) Smoke Sensor Data from the smoke detectors.

2) Heat Sensor Data from the heat detectors.

AO OUTPUTS:

Messages: 1) Emergency Event (to Gas Station Control Queue, loosely-coupled) - indicates that the gas station is on fire.

parameters - none

OBJECT CLASSES REFERENCED:

DETECTOR - encapsulates the Detector STD and provides some operations for sending Emergency Events.

HEAT_DETECTOR - encapsulates raw sensor data from heat detectors.

SMOKE_DETECTOR - encapsulates raw sensor data from smoke detectors.

B) ACTIVE OBJECT STRUCTURE:

Criterion: Control Dependency (Detector object contains STD, encapuslated in the Detector.monitor_sensor operation), Functional Cohesion (Detector object provides operatons to process detector data), and Inheritance (Smoke_Detector and Heat_Detector objects inherit **DETECTOR** object class).

Objects : Detector, Smoke Detector, Heat Detector

C) TIMING CHARACTERISTICS:

Activation: Periodic - activated by a timer, when AO is enabled.

D) PRIORITY:

High - monitors safety conditions.

E) ACTIVE OBJECT EVENT SEQUENCING:

```
Detectors:LINKED_LIST of DETECTOR
NUMBER_OF_DETECTORS:INTEGER is CONSTANT
sd:SMOKE_DETECTOR
hd:HEAT_DETECTOR

Gas_Station_Control:ACTIVE_OBJECT
Timer_Interrupt:AO_INTERRUPT

Create(gsc:ACTIVE_OBJECT, gscq:ACTIVE_QUEUE)
Register:HW_REGISTER

Gas_Station_Control := gsc
Register := DETECTOR_REGISTER_BASE
For NUMBER_OF_DETECTORS
        sd.Create(gscq, Register)
        Detectors.add(sd)
        Register.add(OFFSET)
        hd.Create(gscq, Register)
        Detectors.add(hs)
        Register.add(OFFSET)
End For
Timer_Interrupt.Create(TIMER_INTERRUPT_VECTOR, Poll_Detectors)
end_Create

execute()
        Poll:TIMER

        Poll.Create
        Poll.start(POLL_PERIOD)
        Loop
                Await()
        End  Loop
end_execute

Poll_Detectors
        For Detectors.first until Detectors.last
```

```
            Detectors.item.read
            Detectors.item.process_change(Detectors.item.monitor_sensor)
            Detectors.next
      **End For**
End_Poll_Detectors
```

F) ERRORS DETECTED:

Ignores unrecognized commands.

GAS STATION CONTROL ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: Gas Station Control

A) ACTIVE OBJECT INTERFACE:

AO INPUTS:

Messages: 1) Gas Station Control Queue

a) RCF Event (from Communications Link AO, loosely-coupled) - indicates that a command from the Remote Central Facility has arrived.

parameters - rcf_event (0 = shutdown, 1 = restart)

b) Link Event (from the Communications Link AO, loosely-coupled) - indicates the communications link has changed state

parameters - status (0 = link_down, 1 = link_up)

c) Emergency Event (from Detectors AO, loosely-coupled) - indicates that the gas station is on fire.

parameters - none

AO OUTPUTS:

Messages: 1) Gas Station Event (to Pump Control Queue, loosely-coupled) - requests the pump to change its operating state.

parameters - command (0 = close, 1 = open)

2) Alarm Message (to Transmit Messages Queue, loosely-coupled) - informs the Remote Central Facility about an emergency at the gas station.

parameters - remote address, station_id

Data: 1) Alarm Commands (ON or OFF) - turns the alarm on or off.

OBJECT CLASSES REFERENCED:

GAS_STATION - encapsulates the Gas Station STD and provides some operations for sending messages to Active Objects.

ALARM - encapsulates operations to turn the alarm on and off.

B) ACTIVE OBJECT STRUCTURE:

Criterion:	Control Dependency (Gas Station object contains STD, encapuslated in the Gas Station.process_event operation, that sequences AO operations.), Functional Cohesion (Gas Station object provides operatons to support the STD), and Sequential Cohesion (the Alarm object is used by the Gas Station STD to access alarm functions).
Objects :	Gas_Station, Alarm

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving in the Gas Station Control Queue.

D) PRIORITY:

Medium - lower than the I/O tasks.

E) ACTIVE OBJECT EVENT SEQUENCING:

```
Gas_Station_Control_Queue:ACTIVE_QUEUE

Alarm:ALARM
Gas_Station:GAS_STATION
Gas_Station_Control_Queue:ACTIVE_QUEUE

Create(gscq:ACTIVE_QUEUE, pcql:LINKED_LIST of ACTIVE_QUEUE,
    tmq:ACTIVE_QUEUE, detectors:ACTIVE_OBJECT,
    rcf:ADDRESS, sta_id:INTEGER)

    Gas_Station_Control_Queue := gscq
    Alarm.Create(ALARM_COMMAND_REGISTER)
    Gas_Station.Create(rcf, sta_id, tmq, pcql, detectors, Alarm)
end_Create

execute()
    event:GS_EVENT
    message:MESSAGE
```

```
        known:MESSAGE

        message.Create(SELECTOR)
        Loop
                Wait for message in Gas_Station_Control_Queue
                Switch (message.type)
                        Case RCF Event
                                known.Create(RCF_EVENT, message)
                                if known.rcf_event = shutdown
                                        then event.Create(SHUTDOWN)
                                elsif known.rcf_event = restart
                                        then event .Create(RESTART)
                                fi
                                break
                        Case Link Event
                                known.Create(LINK_EVENT, message)
                                if known.status = link_down
                                        then event.Create(LINK_DOWN)
                                elsif known.status = link_up
                                        then event.Create(LINK_UP)
                                fi
                                break
                        Case Emergency Event
                                event.Create(THRESHOLD_EXCEEDED)
                                break
                End Switch
                Gas_Station.process_event(event)
        End  Loop
end_execute
```

F) ERRORS DETECTED:

Ignores Gas Station STD events that are not recognized.

GAS DISPENSER ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: Gas Dispenser Control

A) ACTIVE OBJECT INTERFACE:

AO INPUTS:

Messages: 1) Gas Command (from Pump Control AO, tightly-coupled, no reply) - requests that the Gas Dispenser start or stop pumping gas.

parameters - command (0 = stop_dispensing, 1 = start_dispensing), Transaction (reference to transaction object).

Data: 1) Meter Data - count of the amount of gas dispensed.

2) Transaction.limit - cash limit on amount of gas to be dispensed

AO OUTPUTS:

Messages: 1) Stopped Event (to Pump Control Queue, loosely-coupled) - indicates that gas is no longer being dispensed.

Data: 1) Dispenser Commands - turns dispenser ON and OFF.

2) Display Data - amount and cost of gas to be displayed for customer viewing.

3) Transaction.cost_of_gas - records the cost of the customer's purchase.

OBJECT CLASSES REFERENCED:

GAS_DISPENSER - encapsulates dispenser operations.

TRANSACTION - encapsulates transaction data.

B) ACTIVE OBJECT STRUCTURE:

Criterion: Asynchronous Event Dependency (gas dispenser is activated and deactivated by events from the Pump Control AO) and Functional Cohesion (the Gas Dispenser Object encapsulates operations supporting gas dispenser functions).

Objects : Gas_Dispenser

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving from the Pump Control Task.

D) PRIORITY:

High - must monitor dispensing operation on a polling basis.

E) ACTIVE OBJECT EVENT SEQUENCING:

```
Gas_Dispenser:GAS_DISPENSER
Pump_Control:ACTIVE_OBJECT
Dispenser_Command:MESSAGE

Create(pc:ACTIVE_OBJECT, pcq: ACTIVE_QUEUE)
    Pump_Control := pc
    Gas_Dispenser.Create(pcq, DISPENSER_REGISTER_BLOCK)
    Dispenser_Command.Create(DISPENSER_COMMAND)
end_Create

execute()
    Transaction:TRANSACTION

    Loop
        Wait for Dispenser_Command from Pump_Control
        if Dispenser_Command.command not = start_dispensing
            then continue
        Transaction := Dispenser_Command.Transaction
        Gas_Dispenser.clear_amount_dispensed
        Gas_Dispenser.start_gas
        Loop
            Gas_Dispenser.update_amount_dispensed
            Gas_Dispenser.update_display
            if Gas_Dispenser.?limit_reached(Transaction)
                then break
            if Waiting Dispenser_Command from Pump_Control
                    then if Dispenser_Command.command = stop_dispensing
                then    break
        End Loop
        Gas_Dispenser.stop_gas(Transaction)
    End Loop
end_execute
```

F) ERRORS DETECTED:

Ignores unrecognized commands.

LED CONTROL ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: LED CONTROL

A) ACTIVE OBJECT INTERFACE:

AO INPUTS:

Messages: 1) Light Command (from Pump Control and Card Reader Control AOs, loosely-coupled) - requests that an LED be lighted.

parameters - LED_ number (0 = Cash_Value_Used, 1 = Cannot_Process_Card)

AO OUTPUTS:

Data: 1) LED Commands - turns LED ON and OFF.

OBJECT CLASSES REFERENCED:

NONE

B) ACTIVE OBJECT STRUCTURE:

Criterion: Resource Monitor Object.

Objects : LED

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving in the Light Queue.

D) PRIORITY:

Medium - controls two LEDs on a pump.

E) ACTIVE OBJECT EVENT SEQUENCING:

```
Light_Queue:ACTIVE_QUEUE
Light_Command:MESSAGE

Create(lq:ACTIVE_QUEUE)
        Light_Queue := lq
        Create.Light_Command(LIGHT_COMMAND)
end_Create

execute()
        Loop
                Wait for Light_Command in Light_Queue
                if Light Command.LED_number = Cash_Value_Used
                        then    output ON to LED_0
                                delay 10 seconds
                                output OFF to LED_0
                elsif Light Command.LED_number = Cannot_Process_Card
                        then    output ON to LED_1
                                delay 10 seconds
                                output OFF to LED_1
                fi
        End Loop
end_execute
```

F) ERRORS DETECTED:

Ignores unrecognized LED numbers.

PUMP CONTROL ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: Pump Control

A) ACTIVE OBJECT INTERFACE:

AO INPUTS:

Messages: 1) Pump Control Queue

a) Switch Event (from Switch AO, loosely-coupled) - indicates that a significant switch event has occurred

parameters - switch_event (0 = switch_off, 1 = switch_on)

b) Card Event (from Card Reader Control AO, loosely-coupled) - indicates that a card was inserted.

parameters - action (0 = cash_card_inserted, 1 = credit_card_inserted), Card (reference to Card Object)

c) Gas Station Event (from Gas Station Control AO, loosely-coupled) - requests that the Pump change its operating state.

parameters - command (0 = close, 1 = open)

d) Authorization Event (from the Communications Link AO or from self, loosely-coupled)

parameters - status (0 = not_okay, 1 = okay)

e) Stopped Event (from Gas Dispenser AO, loosely-coupled) - indicates that gas is no longer being dispensed.

parameters - none

Data: 1) Cash and Credit Card Information from the Card Object (which is created by the Card Reader AO)

2) Transaction Information from the Transaction Object (which is created by Pump Control AO)

ACTIVE OBJECT OUTPUTS:

Messages: 1) Reader Command (to Card Reader Control AO, tightly-coupled, no reply) - requests the Card Reader to execute a command.

parameters - command (0 =Eject, 1 = write), new_cash_value (included only with the write command)

2) Light Command (to Light Queue, loosely-coupled) - requests LED Task light an LED.

parameters - LED_number (0 = Cash_value_used, 1 = Cannot_process_card)

3) Gas Command (to Gas Dispenser AO, tightly-coupled, no reply) - requests that the Gas Dispenser start or stop pumping gas.

parameters - command (0 = stop_dispensing, 1 = start_dispensing), Transaction (reference to transaction object).

4) Authorization Request (to the Transmit Messages Queue, loosely-coupled) - requests that the Remote Central Facility check credit authorization for a customer.

parameters - remote_address, station_id, pump_id, account_number

5) Credit Transaction (to the Transmit Messages Queue, loosely-coupled) - informs the Remote Central Facility about a completed credit transaction.

parameters - remote_address, station_id, pump_id, account_number, amount_purchased

Data: 1) Transaction Object - created by this task.

OBJECT CLASSES REFERENCED:

PUMP - encapsulates the Pump STD

CASH_CARD - encapsulates cash card information read by the Card Reader AO.

CREDIT_CARD - encapsulates credit card information read by the Card Reader AO.

TRANSACTION - encapsulates transaction information and is inherited by **CASH_TRANSACTION** and **CREDIT_TRANSACTION**.
CASH_TRANSACTION - encapsulates transaction data about a cash transaction.
CREDIT_TRANSACTION - encapsulates transaction data about a credit transaction.

B) ACTIVE OBJECT STRUCTURE:

Criterion: Control Dependency (Pump object contains STD, encapuslated in the Pump.process_event operation, that sequences task operations.)

Objects : Pump

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a message arriving in the Pump Control Queue.

D) PRIORITY:

Medium - lower than the I/O tasks.

E) ACTIVE OBJECT EVENT SEQUENCING:

Station_id**:INTEGER**
Pump_id**:INTEGER**
Rcf**:ADDRESS**
Pump **:PUMP**
Pump_Control_Queue**:ACTIVE_QUEUE**
Light_Queue**: ACTIVE_QUEUE**
Card_Reader**: ACTIVE_OBJECT**
Link**:ACTIVE_QUEUE**

Create(pcq**: ACTIVE_QUEUE**, cr**:ACTIVE_OBJECT**, lq**:ACTIVE_QUEUE**,
gd**:ACTIVE_OBJECT**, tmq**:ACTIVE_QUEUE,** rcf**:ADDRESS**
sta_id**:INTEGER,** pump_id**:INTEGER**)

Rcf := rcf
Pump_id := pump_id
Station_id := sta_id
Pump_Control_Queue := pcq
Light_Queue := lq
Card_Reader := cr
Link := tmq

```
Pump.Create(gd, sta_id, pump_id)
end_Create

execute()
    event:PUMP_EVENT
    card: CARD
    message:MESSAGE
    known:MESSAGE
    transaction:TRANSACTION
    cash_tran:CASH_TRANSACTION
    credit_tran:CREDIT_TRANSACTION

    message.Create(SELECTOR)
    Loop
        Wait for message in Pump_Control_Queue
        Switch (message.type)
            Case Switch Event
                known.Create(SWITCH_EVENT, message)
                if known.switch_event = switch_off
                    then event.Create(OFF)
                elsif known.switch_event = switch_on
                    then eventCreate(ON)
                fi
                break
            Case Card Event
                known.Create(CARD_EVENT, message)
                known.card := known.Card
                if known.action = cash_card_inserted
                    then    event.Create(CASH_CARD_INSERTED)
                            transaction :=
                            cash_tran.Create(Pump_Control_Queue,
                                Light_Queue, Card_Reader)
                elsif known.action = credit_card_inserted
                    then    event.Create(CREDIT_CARD_INSERTED)
                            transaction :=
                            credit_tran.Create(Rcf, Station_id, Pump_id
                                Link, Light_Queue, Card_Reader)
                fi
                break
            Case Gas Station Event
                known.Create(GS_EVENT, messsage)
                if known.command = close
                    then event.Create(CLOSE)
                elsif known.command = open
                    then event.Create(OPEN)
                fi
                break
            Case Credit Authorization Event
                known.Create(AUTHORIZATION, message)
                if known.status = not_okay
                    then event.Create(NOT_AUTHORIZED)
                elsif known.status = okay
                    then event.Create(AUTHORIZED)
                fi
                break
            Case Stopped Event
                event.Create(STOPPED)
                break
        End Switch
        Pump.process_event(event, card, transaction)
    End  Loop
end_execute
```

F) ERRORS DETECTED:

Ignores Pump STD events that are not recognized.

SWITCH MONITORING ACTIVE OBJECT BEHAVIOR SPECIFICATION

ACTIVE OBJECT: Switch Monitoring

A) ACTIVE OBJECT INTERFACE:

AO INPUTS:

Events: 1) Switch Activated Interrupt (external event) - indicates pump switch was turned on.
2) Switch Deactivated Interrupt (external event) - indicates pump switch was turned off.

AO OUTPUTS:

Messages: 1) Switch Event (to Pump Control Queue, loosely-coupled) - indicates that a switch event has occurred.

parameters - switch_event (0 = switch_off, 1 = switch_on)

OBJECT CLASSES REFERENCED:

SWITCH - encapsulates Switch STD and some supporting operations.

B) ACTIVE OBJECT STRUCTURE:

Criterion: Control Dependency (Switch STD encapsulated in Switch.process_event operation) and Functional Cohesion (Switch object provides some supporting operations).

Objects : Switch

C) TIMING CHARACTERISTICS:

Activation: Asynchronous by a Switch Interrupt.

D) PRIORITY:

High - captures and records hardware interrupts that would otherwise be lost.

E) ACTIVE OBJECT EVENT SEQUENCING:

Switch**:SWITCH**
Switch_Touch**:AO_INTERRUPT**

```
Create(pcq:ACTIVE_QUEUE)
    Switch.Create(pcq, SWITCH_INPUT_REGISTER)
    Switch_Touch.Create(SW_INTERRUPT_VECTOR, Switch.handle_switch_interrupt)
end_Create

execute()

    Loop
        Await()
    End Loop
end_execute
```

F) ERRORS DETECTED:

None.

AUTOMATED GAS STATION MANAGER

OMT/OODARTS

OBJECT CLASS SPECIFICATIONS

Object Class Specifications

For

Automated Gas Station Manager (AGSM)

Object Class:	**ALARM**
Encapsulates:	Specific method of accessing alarm hardware.
Object Structure:	Device Interface Object
Assumptions:	Sequential assess only.
Anticpated Changes:	None

<u>Attributes</u>

Command_Register**:HW_REGISTER**

<u>Operations</u>

```
Create(cr:HW_REGISTER)
        Command_Register := cr)
end_Create

sound
        output ON_COMMAND to Command_Register
end_sound

reset
        output OFF_COMMAND to Command_Register
end_reset
```

Object Class:	**CARD**
Encapsulates:	Card Identification Number and gives default operations for Account Number and Cash Value stores.
Object Structure:	Data Abstraction Object
Assumptions:	Will be inherited by other objects.

Anticpated Changes:	None

<u>Attributes</u>

identification_number**: NUMERIC**

<u>Operations</u>

```
set_id_number(id:NUMERIC)
        identification_number := id
end_set_id_number

get_id_number:NUMERIC
        return identification_number
end_get_id_number

set_account_number(x:NUMERIC)
        return
end_set_account_number

get_account_number:NUMERIC
        return ZERO
end_get_account_number

set_cash_value(x:INTEGER)
        return
end_set_cash_value

get_cash_value:INTEGER
        return ZERO
end_get_cash_value
```

**Object Class:	CARD_READER**

Encapsulates:	Specifics of interacting with Card Reader hardware.

Object Structure:	Device Interface Object

Assumptions:	Can serve a single task and process one card at a time.

Anticpated Changes:	None

<u>Attributes</u>

```
cc: CASH_CARD
ccc:CREDIT_CARD
Pump_Control_Queue: ACTIVE_QUEUE
Light_Queue: ACTIVE_QUEUE
CR_Base :HW_REGISTER
```

<u>Operations</u>

```
Create(pcq: ACTIVE_QUEUE, lq: ACTIVE_QUEUE, cr_reg:HW_REGISTER)
        Pump_Control_Queue := pcq
        Light_Queue := lq
        CR_Base := cr_reg
end_Create

write(new_cash_value:INTEGER)
        ASCII_value: STRING

        ASCII_value.convert_to( new_cash_value)
        output cc.get_id, ASCII_value to CR_Base+OUT
end_write

Eject
        output EJECT_COMMAND to CR_Base+CMD
end_eject

read:BOOLEAN
        ASCII_ID :STRING
        Card_Event:MESSAGE
        Light_Command:MESSAGE

        input ASCII_ID from CR_Base+IN
        ASCII_ID.convert_from(identification_number)
        if identification_number is for Cash Card
                then    cc.Create
                        cc.set_id(identification_number)
                        input ASCII_value from CR_Base+IN
                        ASCII_value.convert_from(cash_value)
                        cc.set_cash_value(cash_value)
                        Card_Event.Create(CARD_EVENT)
                        Card_Event.action.set(CASH_CARD_INSERTED)
                        Card_Event.Card.set(cc)
                        send Card_Event to Pump_Control_Queue
        eisif identification_number is for Corporate Credit Card
                then    ccc.Create,
                        ccc.set_id(identification_number)
                        input ASCII_value from CR_Base+IN
                        ASCII_value.convert_from(account_value)
                        ccc.set_account_number(account_number)
                        Card_Event.Create(CARD_EVENT)
                        Card_Event.action.set(CREDIT_CARD_INSERTED)
                        Card_Event.Card.set(ccc)
                        send Card_Event to Pump_Control_Queue
        else        Light_Command.Create(LIGHT_COMMAND)
                        Light_Command.LED_number.set(CANNOT_PROCESS_CARD)
                        send Light_Command  to Light_Queue
                        Eject
                        return FALSE
        endif
        return TRUE
end_read
```

Object Class: **CASH_CARD**

Encapsulates: Information store on a Cash Card

Object Structure: Data Abstraction Object

Assumptions: Can be shared between tasks, multiple readers, one writer. Inherits **CARD**.

Anticpated Changes: None

Attributes

cash_value**: INTEGER**

Operations

```
set_cash_value(value:INTEGER)
    cash_value := value
end_set_cash_value

get_cash_value:INTEGER
    return cash_value
end_get_cash_value
```

Object Class: **CASH_TRANSACTION**

Encapsulates: Information about a transaction in progress.

Object Structure: Data Abstraction Object.

Assumptions: Can be shared among tasks, multiple readers, multiple writers. Inherits **TRANSACTION**.

Anticpated Changes: None.

Attributes

Pump_Control_Queue**:ACTIVE_QUEUE**
Light_Queue**:ACTIVE_QUEUE**
Card_Reader**:ACTIVE_OBJECT**

Operations

```
Create(pcq: ACTIVE_QUEUE, lq:ACTIVE_QUEUE, cr:ACTIVE_OBJECT)
        Pump_Control_Queue := pcq
        Light_Queue := lq
        Card_Reader := cr
end_Create

authorize(cc:CARD)
        Auth_Event :MESSAGE

        Auth_Event.Create(AUTHORIZATION)
        if cc.get_cash_value > 0
                then    limit := cc.get_cash_value
                        Auth_Event.status.set(OKAY)
                        send Auth_Event to Pump_Control_Queue
                else    Auth_Event.status.set(NOT_OKAY)
                        send Auth_Event to Pump_Control_Queue
end_authorize

reject
        Light_Command:MESSAGE
        Reader_Command:MESSAGE

        Light_Command.Create(LIGHT_COMMAND)
        Light_Command.LED_number.set(CASH_VALUE_USED)
        send Light_Command to Light Queue
        Reader_Command.Create(READER_COMMAND)
        Reader_Command.command.set(EJECT)
        send Reader_Command  to Card_Reader
end_reject
```

```
complete(cc:CARD)
        Reader_Command:MESSAGE

        new_cash_value := cc.get_cash_value - cost_of_gas
        Reader_Command.Create(READER_COMMAND)
        Reader_Command.command.set(WRITE)
        Reader_Command.data.set(new_cash_value)
        send Reader_Command to Card_Reader
        Reader_Command.Create(READER_COMMAND)
        Reader_Command.command.set(EJECT)
        send Reader_Command to Card_Reader
end_complete
```

Object Class: **CREDIT_CARD**

Encapsulates: Data read from customer's corporate credit card

Object Structure: Data Abstraction Object

Assumptions: Can be shared between tasks, one writer, multiple readers. Inherits **CARD**.

Anticpated Changes: None

Attributes

account_number**: NUMERIC**

Operations

```
set_account_number(account:NUMERIC)
        account_number := account
end_set_account_number

get_account_number:NUMERIC
        return account_number
end_get_account_number
```

Object Class: **CREDIT_TRANSACTION**

Encapsulates: Data associated with a credit transaction that is in progress.

Object Structure:	Data Abstraction Object
Assumptions:	Can be accessed by multiple reader and multiple writer tasks. Inherits **TRANSACTION**.
Anticpated Changes:	None

Attributes

Remote_Central_Facility**: ADDRESS**
Station_ID **: INTEGER**
Pump_ID **: INTEGER**
Link**:ACTIVE_QUEUE**
Light**:ACTIVE_QUEUE**
Card_Reader**:ACTIVE_OBJECT**

Operations

```
Create(rcf:ADDRESS, sid:INTEGER, pid:INTEGER, comm:ACTIVE_QUEUE,
        lq: ACTIVE_QUEUE, cr:ACTIVE_OBJECT)
        Remote_Central_Facility := rcf
        Station_ID := sid
        Pump_ID := pid
        Link := comm
        Light := lq
        Card_Reader := cr
end_create

authorize(ccc:CARD)
        Auth_Request:MESSAGE

        limit : = 0
        Auth_Request.Create(AUTHORIZATION_REQUEST)
        Auth_Request.destination.set(Remote_Central_Facility)
        Auth_Request.station.set(Station_ID)
        Auth_Rqeuest.pump.set(Pump_ID)
        Auth_Request.account.set(ccc.get_account_number)
        send Auth_Request to Link
end_authorize

reject
        Light_Command:MESSAGE
        Reader_Command:MESSAGE

        Light_Command.Create(LIGHT_COMMAND)
```

```
    Light_Command.LED_number.set(CANNOT_PROCESS_CARD)
    send Light_Command to Light
    Reader_Command.Create(READER_COMMAND)
    Reader_Command.command.set(EJECT)
    send Reader_Command to Card_Reader
end_reject

complete(ccc:CARD)
    Credit_Tran:MESSAGE
    Reader_Command:MESSAGE

    Credit_Tran.Create(CREDIT_TRANSACTION)
    Credit_Tran.destination.set(Remote_Central_Facility)
    Credit_Tran.station.set(Station_ID)
    Credit_Tran.pump.set(Pump_ID)
    Credit_Tran.account.set(ccc.get_account_number)
    Credit_Tran.amount.set(cost_of_gas)
    send Credit_Tran to Link
    Reader_Command.Create(READER_COMMAND)
    Reader_Command.command.set(EJECT)
    send Reader_Command to Card_Reader
end_complete
```

Object Class:	**DETECTOR**
Encapsulates:	The abstract logic of a detector device.
Object Structure:	Algorithm Abstraction Object
Assumptions:	Will be inherited by specific types of detector objects.
Anticpated Changes:	None

Attributes

threshold**: REAL is CONSTANT**
status**: INTEGER**
processed_sensor_data**: REAL**
Gas_Station_Control**:ACTIVE_QUEUE**

Operations

```
monitor_sensor:REAL
    delta:REAL
    delta := processed_sensor_data - threshold
    delta.absolute_value
    return delta
end_monitor_sensor

process_change(delta:REAL)
    Encapsulates DETECTOR STD
end_process_change

send_threshold_exceeded
    Emergency_Event:MESSAGE

    Emergency_Event.Create(EMERGENCY)
    send Emergency_Event to Gas_Station_Control
end_send_threshold_exceeded

read          {abstract}
    Must be provided for each type of detector that inherits detector.
end_read
```

Object Class:	**GAS_DISPENSER**
Encapsulates:	Gas dispenser hardware.
Object Structure:	Device Inteface Object
Assumptions:	Executes under control of a sequential task.
Anticpated Changes:	None

Attributes

amount_dispensed**: REAL**
price_per_gallon**: INTEGER is CONSTANT**
Pump_Control **:ACTIVE_OBJECT**
Pump_Control_Queue **:ACTIVE_QUEUE**

Operations

```
Create(pc:ACTIVE_OBJECT, pcq:ACTIVE_QUEUE)
    Pump_Control := pc
    Pump_Control_Queue := pcq
end_Create
```

```
clear_amount_dispensed
        output BLANK_COMMAND to GALLONS_DISPLAY_REG
        output BLANK_COMMAND to COST_DISPLAY_REG
        amount_dispensed := zero
end_clear_amount_dispensed

update_amount_dispensed
        Meter_Data:INTEGER

        input Meter_Data from METER_INPUT_REG
        amount_dispensed := Meter_Data
end_update_amount_dispensed

update_display
        display_gallons:INTEGER
        display_cost:INTEGER

        display_gallons := amount_dispensed/100
        display_cost := display_gallons * price_per_gallon
        output display_gallons to GALLONS_DISPLAY_REG
        output display_cost to COST_DISPLAY_REG
end_update_display

get_amount_dispensed:REAL
        return amount_dispensed
end_get_amount_dispensed

get_price:INTEGER
        return price_per_gallon
end_get_price

?limit_reached(t:TRANSACTION)
        if  t.get_limit = 0 then return FALSE
        if amount_dispensed/100 * price_per_gallon >= t.get_limit
                then return TRUE
                else return FALSE
end_?limit_reached

send_stopped
        Stopped_Event:MESSAGE

        Stopped_Event.Create(STOPPED)
        send Stopped_Event to Pump_Control_Queue
end_send_stopped
```

```
start_gas
    output START_COMMAND to DISPENSER_CNTL_REG
end_start_gas

stop_gas(t:TRANSACTION)

    output STOP_COMMAND to DISPENSER_CNTL_REG
    update_amount_dispensed
    update_display
    t.set_cost_of_gas(amount_dispensed * 100 / price_per_gallon)
    send_stopped
end_stop_gas
```

Object Class:	**GAS_STATION**
Encapsulates:	Gas Station State Transition Diagram
Object Structure:	State Transition Object
Assumptions:	Will execute under control of a single, sequential task.
Anticpated Changes:	None

<u>Attributes</u>

status**: INTEGER**
Pump_Ids**: LINKED_LIST of ACTIVE_QUEUE**
Station_Id**: INTEGER**
Remote_Central_Facility **: ADDRESS**
Alarm**:ALARM**
Link**:ACTIVE_QUEUE**

<u>Operations</u>

```
create( rcf:ADDRESS, sid:INTEGER, comm: ACTIVE_QUEUE, pid_list:LINK_LIST of
    ACTIVE_QUEUE, Whistle:ALARM)

    Link := comm
    Remote_Central_Facility := rcf
    Station_Id := sid
    Pump_Ids := pid_list
    status := OPERATING
    Alarm := Whistle
end_create
```

```
process_event(event:GS_EVENT)
        ENCAPSULATES the Gas Station STATE TRANSITION DIAGRAM
end_process_event

send_alarm_message
        Help:MESSAGE

        Help.Create(ALARM)
        Help.destination.set(Remote_Central_Facility)
        Help.station.set(Station_id)
        send Help to Link
end_send_alarm_message

send_close
        Close:MESSAGE

        for Pump_Ids.first until Pump_Ids.last
                Close.Create(GAS_STATION_EVENT)
                Close.command.set(CLOSE)
                send Close to Pump_Ids.current.item
        end for
end_send_close

send_open
        Open:MESSAGE

        for Pump_Ids.first until Pump_Ids.last
                Open.Create(GAS_STATION_EVENT)
                Open.command.set(OPEN)
                send Open to Pump_Ids.current.item
        end for
end_send_open
```

Object Class:	**HEAT_DETECTOR**
Encapsulates:	Details of specific heat detector hardware.
Object Structure:	Device Interface Module
Assumptions:	Accessed by a single sequential task. Inherits **DETECTOR**.
Anticpated Changes:	None

Attributes

Input_Register**:HW_REGISTER**

Operations

```
Create(gscq:ACTIVE_QUEUE, In_Regsiter:HW_REGISTER)
        Gas_Station_Control := gscq
        status := BELOW_THRESHOLD
        Input_Register := In_Register
end_Create


read
        Sensor_Data: INTEGER

        input Sensor_Data from Input_Register
        processed_sensor_data := Sensor_Data

end_read
```

Object Class:	**LINK**
Encapsulates:	Details associated with communications link hardware.
Object Structure:	Device Interface Object
Assumptions:	Accessible by a sequential task.
Anticpated Changes:	None

Attributes

status**: INTEGER**
RCV_LIST: **LINKED_LIST of MESSAGE**
TX_LIST: **LINKED_LIST of MESSAGE**
CREDIT_TRANSACTION_LIST: **FILE of MESSAGE**
Gas_Station_Control_Queue**:ACTIVE_QUEUE**
Pumps**:ARRAY[ACTIVE_QUEUE]**
Command_Register**:HW_REGISTER**

<u>Operations</u>

```
Create(gscq:ACTIVE_QUEUE, pcql:ARRAY[ACTIVE_QUEUE],
        Cmd_Register:HW_REGISTER)
        status :=  LINK_UP
        Gas_Station_Control_Queue := gscq
        Pumps := pcql
        Command_Register := Cmd_Register
end_Create

add_to_rcv_list(msg:MESSAGE)
        insert msg at tail of RCV_LIST
end_add_to_rcv_list

decode_message
        msg: MESSAGE
        RCF_Event:MESSAGE
        Auth_Event:MESSAGE
        Auth_Reply:MESSAGE

        get msg from head of RCV_LIST
        Switch(msg.type)
                Case   RESTART COMMAND
                        RCF_Event.Create(RCF_EVENT)
                        RCF_Event.rcf_event.set(RESTART)
                        send RCF_Event to Gas_Station_Control_Queue
                        break
                Case SHUTDOWN COMMAND
                        RCF_Event.Create(RCF_EVENT)
                        RCF_Event.rcf_event.set(SHUTDOWN)
                        send RCF_Event to Gas_Station_Control_Queue
                        break
                Case AUTHORIZATION REPLY
                        Auth_Reply.Create(AUTHORIZATION_REPLY,msg)
                        if Auth_Reply.status = OKAY
                                then    Auth_Event.Create(AUTHORIZATION)
                                        Auth_Event.status.set(OKAY)
                                        send Auth_Event to Pumps[Auth_Reply.Pump_id]
                                else    Auth_Event.Create(AUTHORIZATION)
                                        Auth_Event.status.set(NOT_OKAY)
                                        send Auth_Event to Pumps[Auth_Reply.Pump_id]
                        endif
                        break
        End Switch
        remove msg from the RCV LIST
        free msg for use as a new receive buffer
end_decode_message


analyze_link_state(event:LINK_EVENT)
        ENCAPSULATES Link STATE TRANSITION DIAGRAM
end_analyze_link_state

add_to_tx_list(msg:MESSAGE)
        insert msg at tail of TX_LIST
        transmit_message
end_add_to_tx_list

transmit_message
        if TX_LIST is empty then return
        if status is UP  and no message is being transmitted
                then    output TX_CMD, next message from TX_LIST B Command_Register
        elsif status is DOWN
                then save_credit_transactions
        endif
        return
end_transmit_message

save_credit_transactions
        for each message on TX_LIST
                if message is a Credit Transaction
                        then write message to CREDIT_TRANSACTION_LIST
                endif
                remove message from TX_LIST
        end for each message on TX_LIST
end_save_credit_transactions

restore_credit_transactions
        for each message on CREDIT_TRANSACTION_LIST
                add message to head of TX_LIST
                remove message from CREDIT_TRANSACTION_LIST
        end for each message on CREDIT_TRANSACTION_LIST
end_restore_credit_transaction
```

Object Class:	**PUMP**
Encapsulates:	Pump State Transition Diagram
Object Structure:	State Transition Object

Assumptions: Executed under control of a sequential task

Anticpated Changes: None

Attributes

status**: INTEGER**
ID**: INTEGER** -- identity of the Pump
Station**: INTEGER** -- identity of the station where the pump is located
Dispenser**:ACTIVE_OBJECT**

Operations

```
Create(gd:ACTIVE_OBJECT, sta:INTEGER, pid:INTEGER)
    Dispenser := gd
    Station := sta
    ID := pid
    status := OPEN
end_create

process_event(event:PUMP_EVENT, card:CARD, t:TRANSACTION)
    ENCAPSULATES THE PUMP STATE TRANSITION DIAGRAM
end_process_event
```

Object Class: **SMOKE_DETECTOR**

Encapsulates: Details of specific smoke detector hardware.

Object Structure: Device Interface Module

Assumptions: Accessed by a single sequential task. Inherits **DETECTOR**.

Anticpated Changes: None

Attributes

Input_Register**:HW_REGISTER**

Operations

```
Create(gscq:ACTIVE_QUEUE, In_Regsiter:HW_REGISTER)
    Gas_Station_Control := gscq
```

```
        status := BELOW_THRESHOLD
        Input_Register := In_Register
end_Create

read
        Sensor_Data: INTEGER

        input Sensor_Data from Input_Register
        processed_sensor_data := Sensor_Data
end_read
```

Object Class: **SWITCH**

Encapsulates: Switch hardware and state transition diagram.

Object Structure: State Transition Object

Assumptions: Executed by a sequential task

Anticpated Changes: None

Attributes

status**: INTEGER**
Pump_Control_Queue**:ACTIVE_QUEUE**
Switch_Register**:HW_REGISTER**

Operations

```
Create(pcq:ACTIVE_QUEUE, sw_reg:HW_REGISTER)
        Pump_Control_Queue := pcq
        Switch_Register := sw_reg
        status := SWITCH_OFF
end_Create

handle_switch_interrupt
        event:SWITCH_EVENT

        input event from Switch_Register
        process_event(event)
end_handle_switch_interrupt

process_event(event:SWITCH_EVENT)
        ENCAPSULATES SWITCH STATE TRANSITION DIAGRAM
```

```
end_process_event

send_switch_on
        Switch_Event:MESSAGE

        Switch_Event.Create(SWITCH_EVENT)
        Switch_Event.switch_event.set(SWITCH_ON)
        send Switch_Event to Pump_Control_Queue
end_send_switch_on

send_switch_off
        Switch_Event:MESSAGE

        Switch_Event.Create(SWITCH_EVENT)
        Switch_Event.switch_event.set(SWITCH_OFF)
        send Switch_Event to Pump_Control_Queue
end_send_switch_off
```

Object Class: **TRANSACTION**

Encapsulates: Astract concept of a transaction in progress.

Object Structure: Data Abstraction Object

Assumptions: To be inherited by other objects. Can be shared among tasks with multiple readers, multiple writers.

Anticipated Changes: None

<u>Attributes</u>

cost_of_gas**: INTEGER**
limit**: INTEGER**

<u>Operations</u>

```
get_limt:INTEGER
        return limit
end_get_limt

set_cost_of_gas(amount:INTEGER)
        if amount < 0
                then    cost_of_gas := 0
        else
```

```
                    cost_of_gas := amount
          endif
end_set_cost_of_gas

complete            {abstract}
          Must be provided by an inheriting object
end_complete

authorize           {abstract}
          Must be provided by an inheriting object
end_authorize

reject              {abstract}
          Must be provided by an inheriting object
end_reject
```

AUTOMATED GAS STATION MANAGER

OMT/OODARTS

SYSTEM ARCHITECTURE

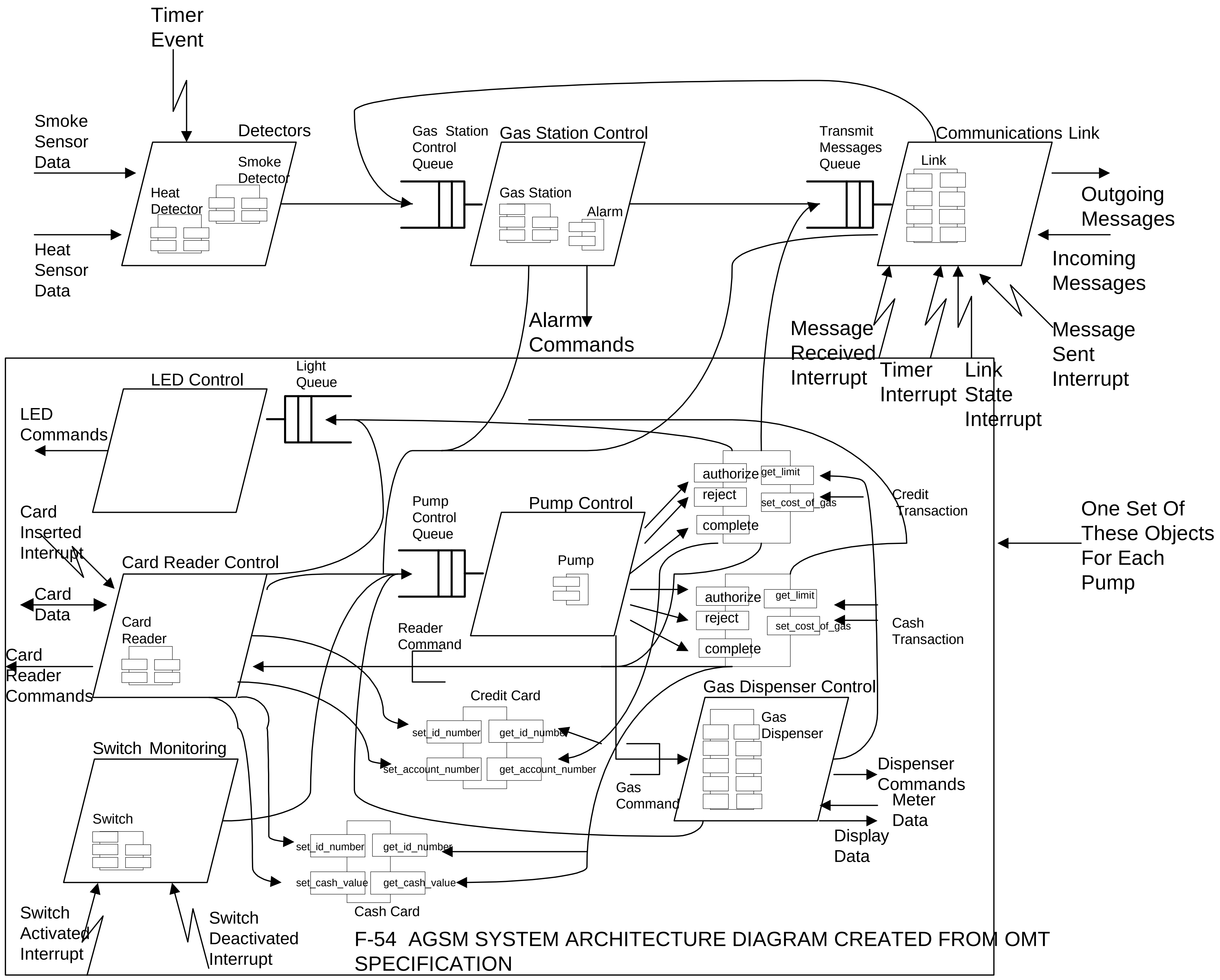
Overview Of AGMS System Architecture

The AGMS system architecture, illustrated in Figure F-54, augments the active object (AO) architecture, shown previously in Figure F-3, by identifying the passive objects within the system and by showing the allocation of those objects among the active objects. A Gas Station object and an Alarm object are included within the Gas Station Control AO. Within the Detectors AO, a Heat Detector object is included for each heat detector in the gas station and a Smoke Detector object is included for each smoke detector. The Communications Link AO includes a Link object.

Turning to the active objects associated with each pump, the Pump Control AO includes a Pump object, the Card Reader Control AO includes a Card Reader object, the Switch Monitoring AO includes a Switch object, and the Gas Dispenser Control AO includes a Gas Dispenser object. (The LED Control AO contains and uses no passive objects.) Within each set of pump AOs, the Cash Card, Credit Card, Cash Transaction, and Credit Transaction passive objects play a key role.

The Cash Card and Credit Card objects are used by the Card Reader Control AO to store the information read from a customer's cash card and credit card, respectively. This information is them accessible to the Cash Transaction or Credit Transaction object, when executing under the thread of control provided by the Pump Control AO. The Cash Transaction object uses the information maintained by the Cash Card object to authorize a transaction and to compute a new cash value. The Credit Transaction object uses the information maintained by the Credit Card object to request, from the remote central facility, authorization of a transaction and to report the completion of a transaction to the remote central facility. The Gas Dispenser Control AO uses the Transaction object (of either type, cash or credit) to determine if a limit is set on the cost of gas to be dispensed and to store the cost of any gas that is dispensed from the gas dispenser. The Pump Control AO uses the Transaction object (of either type) to request that a transaction be authorized, rejected, or completed. These passive objects (Cards and Transaction), shared between active objects, embody the polymorphism described in the OMT specification.

A Cash Transaction object is capable of issuing write and eject commands to the Card Reader Control AO and of issuing light commands to the Light Queue. A Credit Transaction object is capable of issuing eject commands to the Card Reader Control AO, of sending light commands to the Light Queue, and of requesting credit authorization and reporting completed credit transactions via the Transmit Messages Queue.



F-54 AGSM SYSTEM ARCHITECTURE DIAGRAM CREATED FROM OMT SPECIFICATION